División de Ciencias Básicas e Ingeniería

Captura de objetos móviles sobre una recta

Tesis para obtener el grado de

Maestro en Optimización
por
Ing. Luis Eduardo Urbán Rivero

Asesores:
Dr. Rafael López Bracho
Departamento de Sistemas, UAM Azcapotzalco

Dr. Francisco Javier Zaragoza Martínez
Departamento de Sistemas, UAM Azcapotzalco

2 de junio de 2014

Contenido

Lista de Figuras

Lista de Cuadros

Lista de algoritmos

Capítulo 1
Preliminares

1.1. Teoría de gráficas

La teoría de gráficas es un campo de estudio de las matemáticas cuyo objeto principal de estudio son unas estructuras que constan de vértices y aristas que unen estos puntos.

Definición 1.1 *Una* gráfica simple *es un par $G = (V, E)$ donde V es el conjunto de los vértices y E es un conjunto de aristas tal que cada elemento de E es un subconjunto de dos elementos de V.*

Por ejemplo $V = \{1, 2, 3, 4, 5\}$ y $E = \{(1, 2); (2, 3); (3, 1); (5, 4); (5, 1); (3, 4); (4, 2)\}$, cuya representación se puede hacer de manera visual como muestra la Figura 1.1.

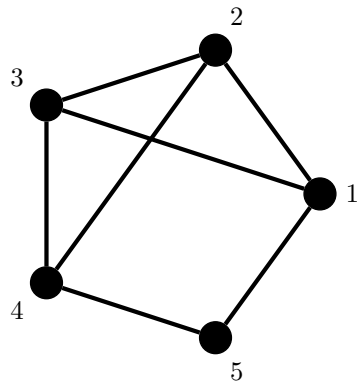

Figura 1.1: Ejemplo de una gráfica simple.

Otro término ampliamente usado es el de subgráfica

Definición 1.2 *Una* subgráfica *S de una gráfica G es una gráfica que cumple las dos condiciones $V(S) \subseteq V(G)$ y $E(S) \subseteq E(G)$.*

Definición 1.3 *Una* subgráfica generadora *de G es una subgráfica S con $V(G) = V(S)$*

De las gráficas se pueden obtener otros tipos de estructuras como las siguientes:

Definición 1.4 *Un* paseo *en G es una sucesión alternada de vértices y aristas de G que inicia y termina en un vértice. Si el paseo termina en el mismo vértice donde inició se considera cerrado; Si termina en uno diferente se considera abierto.*

Definición 1.5 *Un* camino *en G es un paseo en G donde todas las aristas son distintas. Si el camino es cerrado se le conoce como* circuito*.*

Definición 1.6 *Una* trayectoria *en G es un camino en G que no repite vértices, si es cerrado se le conoce como ciclo.*

Definición 1.7 *Un* ciclo hamiltoniano *H de una gráfica G es un ciclo donde $V(H) = V(G)$ y $|E(H)| = |V(G)|$*

Definición 1.8 *Un* camino euleriano *L de una gráfica G es un camino que cumple que $E(L) = E(G)$. Si el camino es cerrado se le llama un* circuito euleriano*.*

Existen también algunas gráficas que por su estructura reciben una clasificación especial.

Definición 1.9 *Un* árbol *T es una gráfica con $|E(T)| = |V(T)| - 1$ y además no contiene ciclos.*

Definición 1.10 *Un* árbol generador *T de una gráfica G es una subgráfica generadora de G que además es un árbol.*

Existen además otro tipo de gráficas que incluyen el concepto de dirección.

Definición 1.11 *Una* gráfica dirigida *es un par $D = (V, A)$ donde V es el conjunto de vértices y A es el conjunto de arcos donde cada elemento de A es un par ordenado de vértices.*

Las gráficas dirigidas también se pueden representar de manera visual. Esta vez se usan flechas para los arcos para representar la dirección, como se muestra en la Figura 1.2.

Definición 1.12 *Un* paseo dirigido *de G es una sucesión alternada de vértices y arcos que inicia y termina en un vértice de G. Si el paseo dirigido termina en el mismo vértice donde inició se considera cerrado; si termina en uno diferente se considera abierto.*

Definición 1.13 *Un* camino dirigido *en G es un paseo dirigido en G donde todas los arcos son distintos, si el camino dirigido es cerrado se le conoce como* circuito dirigido*.*

Definición 1.14 *Una* trayectoria dirigida *en G es un camino dirigido en G que no repite vértices, si es cerrado se le conoce como ciclo dirigido.*

Definición 1.15 *Un* ciclo hamiltoniano dirigido *H de una gráfica dirigida D es un ciclo donde $V(H) = V(D)$.*

Definición 1.16 *Una* gráfica dirigida acíclica *(DAG por sus siglas en inglés) es una gráfica dirigida que no contiene ciclos dirigidos.*

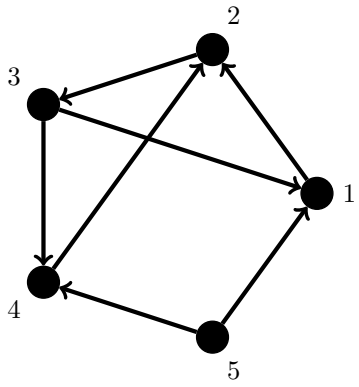

Figura 1.2: Ejemplo de gráfica dirigida

1.2. Complejidad computacional

La teoría de la complejidad computacional es una área de estudio de las ciencias de la computación que estudia el costo involucrado en la solución de los problemas. Se pretende medir la cantidad de recursos a utilizar, como por ejemplo: tiempo, espacio, etc. Una herramienta comúnmente usada en el estudio de la complejidad computacional es el análisis de algoritmos que es el análisis de los recursos usados por un algoritmo dado. Se desean analizar al menos dos aspectos:

- Cota superior: dar un buen algoritmo.
- Cota inferior: demostrar que ningún algoritmo es mejor.

La complejidad computacional además de estudiar el uso de recursos por los algoritmos, también estudia la complejidad de los problemas resueltos por dichos algoritmos, ello para saber en primera si el problema se puede resolver, después saber si va a ser posible encontrar un buen algoritmo o no. Los tipos de problemas que por lo regular se encuentran son los siguientes:

- Problemas triviales.
- Problemas de decisión.
- Problemas de búsqueda.
- Problemas de conteo.
- Problemas de optimización.

En nuestro caso nos concentaremos en lo que son los problemas de decisión y los problemas de optimización.

Definición 1.17 *Un problema P es una relación entre un conjunto I de instancias y un conjunto S de soluciones. Si* $(i, s) \in P$ *diremos que s es una solución para la instancia i.*

Definición 1.18 (Problema de decisión) *Un* problema de decisión *es aquel en el que las instancias se dividen en dos clases: sí y no.*

Definición 1.19 (Problema de optimización) *Un problema de optimización es aquel en el que se desea obtener las mejores soluciones de cada instancia. Cada instancia* $i \in I$ *incluye una función objetivo* f_i *de su conjunto de soluciones* S_i *a un conjunto con un orden total (usualmente los reales) y se desea obtener el menor valor posible de* $f_i(s)$ *con* $s \in S_i$*.*

1.2.1. Algoritmo

Para responder a la pregunta ¿qué es un buen algoritmo? primero se necesita responder, ¿que es un algoritmo? De manera informal se puede ver que un algoritmo es un método efectivo expresado como una lista finita de instrucciones bien definidas. Comenzando desde un estado inicial y con una entrada (posiblemente vacía), las instrucciones describen un proceso determinista que, al ejecutarse, procederá a través de una cantidad finita de estados consecutivos, eventualmente generando una salida (posiblemente vacía) y terminando en un estado final.

Otra forma de definirlo de una manera más formal es.

Definición 1.20 *Un algoritmo A es una función de su conjunto de entradas I a su conjunto de salidas S. Si se ejecuta A con la entrada* $e \in I$ *y al terminar produce la salida* $s \in S$*, diremos que* $A(e) = s$*.*

Además de esta definición los algoritmos que necesitaremos se deben asociar a la resolución de un problema dado.

Definición 1.21 *Diremos que un algoritmo* $A : I \to S$ *resuelve un problema* $P : I \to S$ *si* $A(e) = P(e)$ *para toda* $e \in I$

De manera informal podemos decir que un *buen algoritmo*, es aquel que se ejecuta en un tiempo razonable (se puede realizar en un tiempo humanamente viable). Además se necesita poder expresar la cantidad de recursos utilizados por un algoritmo como una función, dichas funciones dan origen a lo que se conoce como clases de complejidad, por ejemplo:

1. Logarítmicas $\log_2 n$.
2. Lineales $3n + 5$.
3. Polinomiales $n^3 + 2n^2 + 3$.

4. Exponenciales 3^n.

5. Factoriales $n!$.

De aquí podemos decir que del 1 al 3 de la lista se consideran buenos algoritmos y el 4 y 5 son malos algoritmos en lo que respecta al uso de recursos.

1.2.2. Notación O, Ω y Θ

Para poder explicar esto con mayor precisión (sin depender de factores constantes, como el lenguaje de programación, la habilidad del programador o la potencia del equipo donde se ejecuten los algoritmos), se requiere de una medida de complejidad que mida el uso de los recursos en función del tamaño de su entrada. Para ello requeriremos de las siguientes notaciones que ignoran dichos factores.

Definición 1.22 *Diremos que $f(n) \in O(g(n))$ si existen constantes $c > 0$ y $n_0 \geq 0$ tales que $f(n) \leq cg(n)$ para toda $n \geq n_0$. .*

Definición 1.23 *Diremos que $f(n) \in \Omega(g(n))$ si existen constantes $c > 0$ y $n_0 \geq 0$ tales que $f(n) \geq cg(n)$ para toda $n \geq n_0$.*

Definición 1.24 *Diremos que $f(n) \in \Theta(g(n))$ si $f(n) \in O(g(n))$ y $f(n) \in \Omega(g(n))$.*

Estas notaciones a su vez cumplen con ciertas propiedades algebraicas.

Teorema 1.1 (Primer teorema de la suma) *Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$ entonces $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$.*

Teorema 1.2 (Segundo teorema de la suma) *Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$ entonces $f_1(n) + f_2(n) \in O(\text{máx}(g_1(n), g_2(n)))$.*

Teorema 1.3 (Teorema del producto) *Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$ entonces $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$.*

1.2.3. Problemas P y NP

Como ya se había mencionado antes, los algoritmos resuelven problemas. Entonces se tiene un algoritmo con cierto orden de ejecución y que resuelve un problema dado.

- Constante $O(1)$: determinar si un número es par.
- Logaritmico $O(\log n)$: búsqueda binaria.
- Lineal $O(n)$: suma de n bits.

- Polinomial $O(n^c)$: acoplamiento máximo.
- Exponencial $O(c^n)$: generación de cadenas c-arias.
- Factorial $O(n!)$: generación de permutaciones.

Los anteriores son los ordenes que corresponden a cierto algoritmo que resuelve el problema mencionado, sin embargo eso no denota la complejidad del problema que resuelven. Para denotar la complejidad de los problemas son necesarios otros conceptos.

Anteriormente se mencionó que las notaciones O, Ω y Θ son funciones del tamaño de la entrada, pero para que ello tenga sentido debemos mencionar cómo se representa dicha entrada. Una convención ampliamente utilizada es usar la codificación que requiera la menor cantidad de símbolos (bits) para representar a los objetos correspondientes.

Existen algoritmos que por sus características pueden clasificarse en las siguientes clases.

Definición 1.25 *Un algoritmo se dice* polinomial *si su tiempo de ejecución es $O(p(n))$ para algún polinomio p en el tamaño n de su entrada.*

Con respecto a los problemas que resuelven los algoritmos polinomiales tenemos la siguiente definición.

Definición 1.26 *Un problema de decisión se dice* polinomial *si existe algún algoritmo polinomial que lo resuelva. El conjunto de estos problemas es la clase P.*

Definición 1.27 *Un algoritmo no determinista se dice* polinomial *si su tiempo de ejecución es $O(p(n))$ para algún polinomio p en el tamaño n de su entrada.*

Definición 1.28 *Un problema de decisión se dice* polinomial no determinista *si existe algún algoritmo polinomial no determinista que lo resuelva. El conjunto de estos problemas es la clase NP.*

El término no determinista, está asociado a lo que se conoce como una máquina de Turing no determinista. Para entender las implicaciones que esto conlleva solo se mostrara la diferencia entre una máquina de Turing determinista y la no determinista.

Definición 1.29 *Una máquina de Turing T determinista acepta una entrada e si la ejecución de $T(e)$ lleva a T a un estado final $f \in F$, donde F es el conjunto de estados finales.*

Definición 1.30 *Una máquina de Turing T no determinista acepta una entrada e si alguna ejecución de $T(e)$ lleva a T a un estado final $f \in F$ donde F es el conjunto de estados finales.*

La diferencia parece ser muy sutil, sin embargo la máquina no determinista implica averiguar cual es la ejecución que implicó el estado final.

La división de los problemas en las clases anteriores nos lleva a plantearnos las siguientes clases para problemas de un tipo en particular.

Definición 1.31 *Un problema de decisión D se dice* NP completo *si:*

- *Está en NP.*
- *Todo problema en NP se puede reducir a D en tiempo polinomial.*

Definición 1.32 *Un problema D es* NP duro *si todo problema en NP se puede reducir a D en tiempo polinomial.*

Existen una gran variedad de problemas que caen en esta clase. Para profundizar más en el tema se recomienda verificar [9].

Adicionalmente cabe mencionar que saber si $NP = P$ es un problema abierto, considerado uno de los Problemas del Milenio [**?**].

1.2.4. Problemas NP de optimización

Otra clase de problemas involucrada en este trabajo es la que tiene que ver con los problemas NP de optimización.

Definición 1.33 *Un problema de NP optimización* $OPT : I \to S$ *cumple que:*

1. *Su conjunto I de instancias se puede reconocer en tiempo polinomial (es decir, el problema de decisión* $e \in I$ *está en P).*
2. *Si* $e \in I$ *y* $S(e)$ *es su conjunto de soluciones entonces:*
   a) *Existe un polinomio p tal que si* $s \in S(e)$ *entonces* $|s| \leq p(|e|)$.
   b) *Si* $|t| \leq p(|e|)$, *se puede decidir en tiempo polinomial si* $t \in S(e)$.
3. *La función objetivo* $m : I \times S \to \mathbb{Z}$ *se puede calcular en tiempo polinomial.*

De lo anterior podemos definir la clase NPO para todos los problemas NP de optimización. Además ahora podemos decir que si tenemos un problema $OPT \in NPO$ que se puede resolver en tiempo polinomial. Entonces su versión de decisión D_{OPT} también se puede resolver en tiempo polinomial. En conclusión, si $P \neq NP$ y D_{OPT} es NP completo, entonces OPT no se puede resolver en tiempo polinomial.

1.3. Algoritmos de aproximación

Si tenemos un problema de optimización y estamos dispuestos a sacrificar la optimalidad y buscar soluciones aproximadas que se puedan obtener en tiempo polinomial, entonces una posibilidad es recurrir a los algoritmos de aproximación. Para definir qué es un algoritmo de aproximación primero debemos definir el término desempeño.

Definición 1.34 *Sea $e \in I$ y $s \in S(e)$. Entonces, el desempeño de s con respecto a e está dado por*

$$R(e,s) = \frac{m(e,s)}{OPT(e)},$$

es decir, la razón entre el valor de s y el valor de una solución óptima.

Note que $R(e,s) \leq 1$ en un problema de maximización y $R(e,s) \geq 1$ en un problema de minimización. Del concepto de desempeño podemos derivar el termino garantía.

Definición 1.35 *Sea $OPT \in NPO$ de minimización, sea A un algoritmo que para toda $e \in I$ regresa $A(e) \in S(e)$ y sea $\alpha : \mathbb{N} \to \mathbb{R}^+$. Si A cumple que*

$$R(e, A(e)) \leq \alpha(|e|),$$

para toda $e \in I$, entonces diremos que A es un algoritmo de aproximación para OPT con garantía α. Si además A se ejecuta en tiempo polinomial, diremos que OPT se puede aproximar con garantía α.

Para $OPT \in NPO$ de maximización se cambia la desigualdad por $R(e, A(e)) \geq \alpha(|e|)$. Para el caso en que α sea constante se dice que el problema $OPT \in NPO$ pertenece a la clase APX. En el caso de que α no sea constante, se puede presentar lo que se conoce como esquema de aproximación.

1.4. Programación lineal

La programación matemática y en particular la programación lineal son herramientas generales de resolución de problemas de problemas de optimización. En algunos textos [15] la califican junto con la programación dinámica, como un cañón para matar moscas, esto denota el potencial de esta herramienta en el modelado de problemas de optimización.

Un programa lineal consta de dos partes principalmente. La primera es una función objetivo y por otro lado un conjunto de restricciones. La forma canónica es la siguiente:

$$\begin{array}{c} \text{máx } cx \\ \text{sujeto a:} \\ Ax \leq b \\ x \geq 0, \end{array} \tag{1.1}$$

en donde c es un vector de costos, x es un vector de variables de decisión, A es la matriz de coeficientes de las restricciones y b es el lado derecho de las restricciones, la última línea del programa 1.1 denota que se utilizan variables no negativas, sin embargo utilizar variables con signo es fácilmente adaptable.

Sabiendo que la programación lineal nos sirve como herramienta de modelado para la resolución de problemas, ahora nos resta decir cómo se resuelve dicho programa lineal.

1.4.1. Método simplex

En 1947 Dantzig propone un método de solución de programas lineales, conocido como método simplex, en sus inicios este método tuvo aplicaciones militares, por lo que no fue publicado [15]. Este método, se basa en los siguientes resultados de Nelder y Mead [**?**]:

Teorema 1.4 *El conjunto de soluciones factibles de un programa lineal es convexo.*

Teorema 1.5 *La función objetivo de un programa lineal tiene su valor óptimo (máximo o mínimo), en un punto extremo (vértice) del conjunto convexo de soluciones factibles.*

Teorema 1.6 *Una condición necesaria y suficiente para que un punto $x \geq 0$ en el conjunto de soluciones factibles sea punto extremo, es que x sea una solución básica factible que satisfaga el sistema: $Ax = b$.*

El sistema $Ax = b$ proviene de la forma estándar de un programa lineal, en donde se agregan variables de holgura para que el programa lineal sea un sistema de ecuaciones. En este contexto una solución básica es aquella que es solución de ese sistema de ecuaciones.

El método simplex es un procedimiento que explora los puntos extremos del conjunto convexo de soluciones factibles hasta encontrar el óptimo. En la mayoría de los casos, este procedimiento no es exhaustivo en el conjunto de puntos extremos. Sin embargo hay ejemplos que obligan a que el simplex se comporte como un algoritmo exhaustivo [13]. De comportarse como un algoritmo exhaustivo su tiempo de ejecución es exponencial en el número de variables y restricciones.

1.4.2. Algoritmos de punto interior

Como ya mencionamos el método simplex puede tener un tiempo de ejecución exponencial aunque no sea para la mayoría de los casos. Para evitar este inconveniente se han propuesto diversos algoritmos que se basan en la idea de tomar una solución factible del conjunto convexo y a partir de eso obtener un conjunto convexo más pequeño que mantenga la solución óptima. El primero de estos algoritmos fue propuesto en [11], se conoce como el algoritmo elipsoidal y su tiempo de ejecución es de $O(p^{6.5}(|A| + |b| + |c|)^2)$ donde p es el numero de variables. Una mejora a este algoritmo se puede encontrar en [12] con un algoritmo de orden $O(p^{3.5}(|A| + |b| + |c|)^2)$.

Nota:Los algoritmos deben considerar el valor de la entrada y la representación que tenga esta en el tiempo de ejecución. Por lo cual estos algoritmos no son completamente polinomiales y les conoce como algoritmos Pseudopolinomiales.

Encontrar un algoritmo que resuelva programación lineal en tiempo polinomial, sin depender de la entrada, es aun un problema abierto [**?**].

1.5. Programación dinámica

La programación dinámica es un método general de resolución de problemas propuesto en [5], basado en la idea de divide y vencerás, es decir, en la descomposición del problema en subproblemas más simples. Los problemas a los que esta herramienta se puede aplicar, son aquellos en donde la solución de los subproblemas se conserva en el problema del cual vienen y se puede utilizar en la solución de dicho problema.

1.5.1. Funciones recursivas

Una forma de comprender el funcionamiento de la programación dinámica, es teniendo como antecedente a las funciones recursivas donde el problema es también dividido en subproblemas, pero en donde no se reutilizan las soluciones de los subproblemas, si no que se recalculan cada vez que se necesitan. Para poder visualizar esto se muestra el siguiente ejemplo.

Para calcular el n-ésimo número de Fibonacci se usa la función recursiva

$$Fib(n) = \begin{cases} 1 & \text{si} \quad n = 1 \text{ o } 0 \\ Fib(n-1) + Fib(n-2) & \text{en otro caso.} \end{cases} \tag{1.2}$$

Ahora veamos que si quiero calcular $Fib(5)$ se requiere calcular lo mostrado en la Figura 1.3.

Como se puede observar, se requiere repetir muchas veces el mismo cálculo, esto en un ejemplo pequeño no tiene mucho significado pero en cálculos de números más grandes comienza a complicarse. El tiempo de ejecución para calcular la Función 1.2 es :

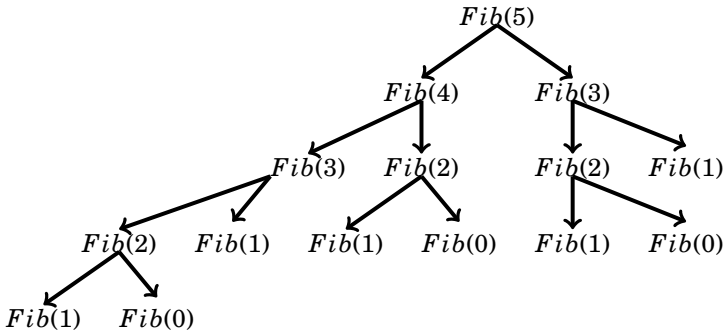

Figura 1.3: Calculo de $Fib(5)$

$$T(n) = \begin{cases} 1 & \text{si} \quad n = 1 \text{ ó } 0 \\ T(n-1) + T(n-2) + 1 & \text{en otro caso.} \end{cases} \tag{1.3}$$

Al resolver la relación de recurrencia obtenemos que el algoritmo basado en la función recursiva 1.2 es de orden $O(\phi^n)$ donde $\phi = \frac{1+\sqrt{5}}{2} \approx 1.6$. Lo anterior supone un alto costo computacional para un problema sumamente sencillo, en este punto es donde interviene la programación dinámica. Se necesitará un arreglo lineal de tamaño $n+1$ que denotaremos como F, con $F[0] = 1$ y $F[1] = 1$. Si queremos calcular $F[n]$ bastará con seguir el Algoritmo 1.1:

Algoritmo 1.1 Números de Fibonacci con programación dinámica

Entrada: n
Salida: $F[n]$
1: $F[0] = 1$.
2: $F[1] = 1$.
3: **para** $i = 2$ **hasta** n **hacer**
4: $\quad F[i] = F[i-1] + F[i-2]$.
5: **fin para**
6: **devolver** $F[n]$

El Algoritmo 1.1 solo necesita recorrer el arreglo hasta la posición n y generar las $n-1$ posiciones previas por lo que el tiempo de ejecución de este algoritmo es de $O(n)$, lo cual es considerablemente menor que $O(\phi^n)$

Note que en este caso no se llama a una función, si no que se toma un valor previamente almacenado. En la Figura 1.4 se puede ver el calculo de $F[5]$ con el algoritmo 1.1.

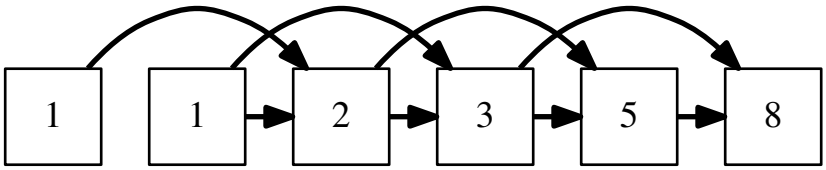

Figura 1.4: Cálculo de $F[5]$

1.5.2. Problema de caminos más cortos

Otro problema que resultará de interés para este trabajo, es el problema de caminos más cortos (SPP por sus siglas en inglés). En dicho problema se tiene una gráfica simple $G = (V, E)$ y una función de costo $c : E \to \mathbb{Z}^+$, y se quiere encontrar el camino de menor costo entre un par de vértices $u, v \in V$. Note que si todos los costos de las aristas son 1 entonces el problema es encontrar el camino más corto entre los vértices u y v.

El algoritmo de Floyd-Warshall emplea las ideas de la programación dinámica en la obtención de los caminos más cortos en una gráfica **[?]**.

Algoritmo 1.2 Algoritmo de Floyd-Warshall

Entrada: Gráfica $G = (V, E)$, $c : E \to \mathbb{Z}^+$
Salida: Longitud de caminos más cortos entre cada par de vértices $u, v \in V$

1: **para cada** $e = uv \in E$ **hacer**
2: $\quad camino[u, v] = c[u, v]$.
3: **fin para**
4: **para** $k = 0$ **hasta** n **hacer**
5: $\quad$ **para cada** $e = uv \in E$ **hacer**
6: $\quad\quad camino[u, v] = \min(camino[u, v], camino[u, k] + camino[k, v])$.
7: $\quad$ **fin para**
8: **fin para**
9: **devolver** camino

El Algoritmo 1.2 es de orden $O(n^3)$ debido a que para cada par de vértices se debe verificar si existe un camino más corto que involucre algún vértice intermedio $1, 2, \ldots, k$.

Como se puede ver en los dos ejemplos anteriores, la programación dinámica puede ayudar a reducir el tiempo de ejecución de un algoritmo de manera considerable (en ocasiones de exponencial a polinomial).

Capítulo 2
Problemas tipo agente viajero

El problema del agente viajero (TSP por sus siglas en inglés) se puede considerar uno de los problemas clásicos de la optimización combinatoria. Desde el punto de vista informal podemos ubicar a principios de los años 60 un concurso de la empresa Procter & Gamble, en dicho concurso se ofrecía un premio de 10,000 dólares [1] a quienes resolvieran una instancia de este problema con 33 ciudades. Más allá de las anécdotas que surjan, existen extensos estudios teóricos y diversas aplicaciones, en ellas puede que no se utilice el TSP en su forma general, puesto que no refleja algo cercano a la realidad. Sin embargo seguramente sí se utilizan algunas variantes o versiones modificadas del problema para adaptarlas al entorno de aplicación. Por ejemplo, las empresas continuamente tienen que repartir sus mercancías, para repartir dichas mercancías se que requiere en la mayoría de los casos, más de un vehículo, los vehículos tienen una capacidad de carga limitada y su uso está en función de la demanda de los clientes. A esta versión se le conoce como el problema de rutas vehiculares (VRP por sus siglas en inglés). Sin embargo ese problema no ocupará nuestra atención, pero es una buena forma de ejemplificar el argumento anterior.

2.1. TSP en general

En el TSP se tiene una gráfica simple $G = (V, E)$ donde V es el conjunto de vértices y E el conjunto de aristas además de una función $c : E \to \mathbb{Q}^+$ y se busca encontrar el ciclo hamiltoniano de costo mínimo. Lo primero que se debe cumplir antes de intentar resolver el TSP es que la gráfica debe tener al menos un ciclo hamiltoniano, para simplificar trabajaremos sobre una gráfica G' que es una gráfica completa que tiene los mismos vértices y aristas de G más las aristas que no tenía G, función de costo en estas nuevas aristas se le asigna un valor M que es un valor muy grande. Si el ciclo hamiltoniano encontrado usó alguno de estos valores M entonces la gráfica original no tenía un ciclo hamiltoniano.

Pese a la apariencia simple de este problema, la realidad es que en el sentido de complejidad computacional es un problema NP-Duro [9], es decir que a menos de que

P=NP, no existe un algoritmo que resuelva el problema de manera exacta en un tiempo razonable (polinomial). Podemos darnos una idea de ello explorando algunos de los algoritmos que existen para resolver el TSP de manera exacta. Dado que necesitamos encontrar un ciclo que visite todos los vértices y regrese al vértice donde inició, Se puede representar el recorrido como una secuencia de vértices, que contiene una permutación de todos los vértices más el vértice inicial repetido al final de la secuencia. Por ejemplo, para una gráfica de 5 vértices donde el vértice inicial repetido al final para cerrar el ciclo, si tenemos la siguiente secuencia $1 \to 5 \to 3 \to 4 \to 2 \to 1$ un primer algoritmo sería tomar la permutación y sumar los valores de las aristas representadas, repetir esto con cada permutación y tomar la que nos dé el valor más pequeño; esto nos llevará $\frac{n!}{2}$ pasos siendo n la cantidad de vértices de la gráfica, la división entre 2 es por las secuencias que son simétricas. Para darse una idea de cual sería el tiempo necesario: si se desea calcular una instancia de apenas 33 vértices tendremos que evaluar 4×10^{36} pasos. Suponiendo que cada paso se lleva 1 nanosegundo la tarea completa necesitará aproximadamente 10^9 siglos. Lo cual se ve claramente superado con cualquier problema de la industria, donde podemos encontrar vehículos que visitan alrededor de 100 lugares en un recorrido.

2.1.1. Solución basada en programación dinámica

Una de las metodologías que primero se emplearon para manejar la complejidad del TSP, fue el uso de la programación dinámica. Dicha técnica se basa en la construcción parcial de un ciclo que garantice su optimalidad para ciclos más pequeños que n y aumentándolo hasta alcanzar un ciclo de tamaño n. En el algoritmo será necesario explorar todos los subconjuntos de un conjunto de tamaño n con lo que se requerirán 2^n pasos más lo necesario para que se complete el ciclo, es decir, se exploran todas las combinaciones de 2 elementos, lo que se traduce en n^2 pasos para cada camino creado con este método con lo que nos da un total de $n^2 \cdot 2^n$ pasos. La solución está dada en forma de una función recursiva $C(S, j)$, que es la mínima longitud del camino que va desde el vértice 1 pasando por todos los vértices del conjunto S y llega al vértice j. Se puede expresar como sigue:

$$C(S,k) = \begin{cases} d_{1,m} & \text{si} \quad S = \{1,k\} \\ \text{mín}_{m \neq k, m \in S}\{C(S - \{k\}, m) + d_{m,k}\} & \text{en otro caso} \end{cases} \tag{2.1}$$

Donde $d_{m,k}$ es la distancia entre un vértice m y un vértice k.

2.1.2. Solución basada en programación matemática

Otra forma de ver el TSP es mediante el uso de la programación matemática, en particular la programación entera. Para ello primero definiremos las variables de decisión $x_{i,j}$

donde $x_{i,j} = 1$ si se utiliza la arista ij y $x_{i,j} = 0$ en otro caso. Posteriormente definiremos una función objetivo:

$$z = \text{mín} \sum_{i \in V, j \in V, i \neq j} c_{i,j} x_{i,j} \tag{2.2}$$

Posteriormente plantearemos un conjunto de restricciones que nos permitan formar un ciclo y visitar todos los vértices. Se puede observar que en un ciclo cada vértice tiene una arista de entrada y una de salida, de aquí las siguientes restricciones:

$$\sum_{j \in V} x_{i,j} = 1 \quad \forall i \tag{2.3}$$

$$\sum_{i \in V} x_{i,j} = 1 \quad \forall j \tag{2.4}$$

Lo anterior es insuficiente para representar el problema puesto que se pueden formar subciclos sin violar ninguna restricción.

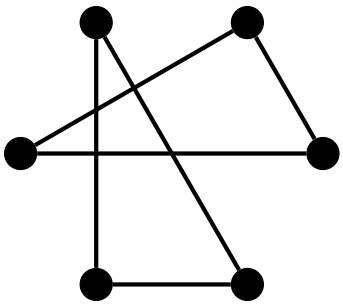

Figura 2.1: Ejemplo donde se forman subciclos y no se viola ninguna restricción puesto que para cada vértice sale y entra una arista.

Existen diversas formas de eliminar la existencia de subciclos, sin embargo, sólo utilizaremos una que involucra el manejo de subconjuntos.

$$\sum_{i,j \in S} x_{i,j} = |S| - 1 \quad \forall |S| = 2, 3, \ldots, |V| - 1 \tag{2.5}$$

Con $S \subseteq V$ además de las restricciones $0 \leq x_{i,j} \leq 1$.

La formulación anterior fue propuesta en [7]. Podría parecer que el problema ha quedado resuelto, sin embargo el número de restricciones necesarias son del orden $O(2^n)$ así que tan solo el hecho de generar las restricciones nos ocupa tiempo exponencial, además del tiempo que se necesite para resolver el modelo. Lo importante de esta formulación es hacer notar que es posible modelar y resolver este problema con ayuda de la programación matemática.

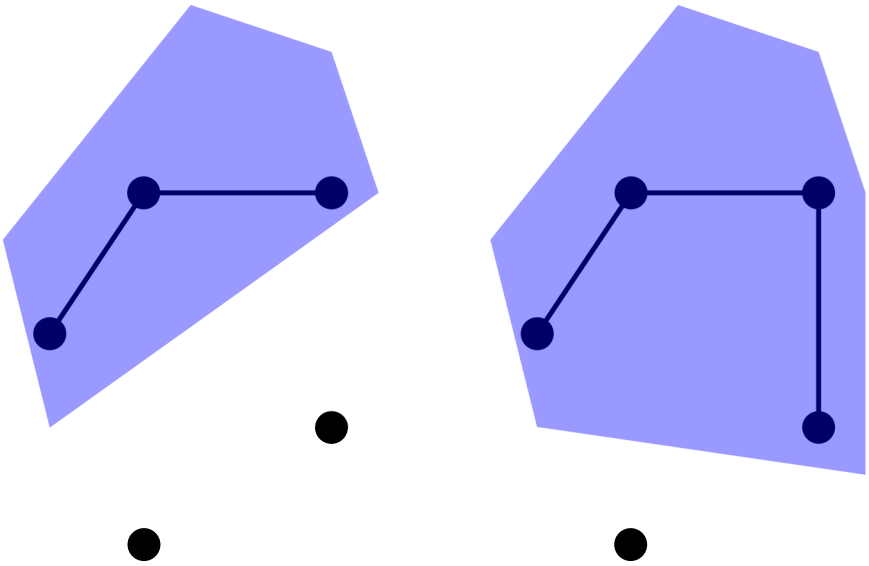

Figura 2.2: Ejemplo del incremento del conjunto S (azul), a la izquierda $|S| = 3$ y a la derecha $|S| = 4$, nótese que el número de aristas es $|S| - 1$ tal y como lo muestra el modelo

2.2. TSP métrico

Pareciera ser que no existe la posibilidad de mejorar el panorama anterior, sin embargo se pueden agregar algunas restricciones al problema del TSP y con ello tomar algunas propiedades de estas restricciones que permiten su solución aproximada. Una restricción adicional surge de la necesidad de describir un problema real donde es posible trasladarse desde un vértice i a un vértice j, en donde es más económico hacerlo de manera directa que a través de vértices intermedios, por lo tanto la función de costo debe cumplir las siguientes propiedades:

1. $c(i,i) = 0$ para cualquier $i \in V$.

2. $c(i,j) = c(j,i) \geq 0$ para cualquier $i, j \in V$.

3. $c(i,k) \leq c(i,j) + c(j,k)$ para cualquier i, j y $k \in V$.

Esta serie de propiedades nos dicen que quedarme en un vértice no aumenta el costo de la solución, la segunda que el sentido en que se recorra la arista no importa y la tercera conocida como la desigualdad del triángulo dice que si hay un camino para ir de un vértice i a un vértice k con un camino intermedio, el camino directo siempre va a tener un costo igual o menor.

2.2.1. Algoritmo basado en árboles de máxima expansión de costo mínimo

El siguiente algoritmo se basa en la construcción de un árbol de expansión de costo mínimo T de una gráfica $G = (V, E)$. Este árbol T es una subgráfica generadora de la gráfica G, con la propiedad de que su número de aristas es $|V| - 1$ lo cual nos garantiza que no tiene ciclos pero que conecta a todos los vértices de G.

Algoritmo 2.1 Algoritmo para TSP métrico con factor de aproximación 2

Entrada: Gráfica $G = (V, E)$ con propiedades métricas en los costos.
Salida: Ciclo hamiltoniano cuyo costo tiene un factor 2 de aproximación
1: Buscar un árbol de máxima expansión T, de peso mínimo
2: Duplicar cada arista de T para obtener una gráfica euleriana L
3: Buscar un circuito euleriano ϕ en la gráfica L
4: Crear un ciclo C que visite los vértices de G en el orden en que aparecen por primera vez en ϕ
5: **devolver** C

El algoritmo 2.1 genera una solución factible para el TSP métrico. En la línea 4 es donde se da la creación del ciclo hamiltoniano, es aquí donde se aprovecha la desigualdad del triángulo debido a que si se llega a algún vértice que ya tenía construido un camino por la gráfica L y existe un camino directo este será intercambiado y debido a la desigualdad del triángulo su costo no aumentará, en el peor de los casos quedará igual, además si el vértice ya fue visitado evitará formar ciclos excepto para el vértice donde comenzó el recorrido cuando ya se hayan visitado todos los vértices de la gráfica G.

Teorema 2.1 *El algoritmo 2.1 tiene una garantía de aproximación 2*

Demostración. Sea OPT el ciclo hamiltoniano óptimo. Se sabe que $costo(T) \leq costo(OPT)$ además de que la gráfica ϕ contiene por duplicado las aristas de T asi que $costo(\phi) = 2\, costo(T)$ y gracias a la propiedad de la desigualdad del triángulo se sabe que $costo(C) \leq costo(\phi)$, por lo tanto.

$$costo(OPT) \leq costo(C) \leq 2\, costo(OPT). \tag{2.6}$$

2.2.2. Algoritmo de Christofides

Es posible mejorar la garantía del algoritmo anterior, con unos cuantos cambios se puede mejorar el factor de aproximación de 2 a $\frac{3}{2}$.

El algoritmo 2.2 integra el concepto de acoplamiento perfecto de costo mínimo para poder realizarse. El acoplamiento es lo que permite mejorar la cota de aproximación del algoritmo.

Algoritmo 2.2 Algoritmo para TSP métrico con factor de aproximación $\frac{3}{2}$

Entrada: Gráfica $G = (V, E)$ con propiedades métricas en los costos.
Salida: Ciclo hamiltoniano cuyo costo tiene un factor $\frac{3}{2}$ de aproximación

1: Buscar un árbol de máxima expansión T, de peso mínimo
2: Calcular el acoplamiento perfecto de costo mínimo M, en el conjunto de vértices con grado impar de T. Agregar M a T y obtener una gráfica euleriana L
3: Buscar un circuito euleriano ϕ en la gráfica L
4: Crear un camino C que visite los vértices de G en el orden en que aparecen por primera vez en ϕ
5: **devolver** C

Lema 2.1 (Christofides, 1976) *Sea $V' \subseteq V$ tal que $|V'|$ es par, y sea M un acoplamiento perfecto de costo mínimo en V'. Entonces $costo(M) \leq costo(OPT)/2$.*

Demostración. Considere un ciclo hamiltoniano óptimo τ de G. Sea τ' un ciclo sobre V' obtenido de corto circuitar a τ, por la desigualdad de l triangulo $costo(\tau') \leq costo(\tau)$. Además de que τ' esta formado por la unión de dos acoplamientos perfectos en V', el mas barato de estos acoplamientos tiene costo $costo(\tau')/2 \leq costo(OPT)/2$. ■

Teorema 2.2 (Christofides, 1976) *El algoritmo 2.2 tiene un factor de aproximación de $\frac{3}{2}$.*

Demostración. A partir del Lema 2.1 se deduce lo siguiente:

$$costo(\phi) \leq costo(T) + costo(M) \leq costo(OPT) + \frac{1}{2}costo(OPT) = \frac{3}{2}costo(OPT). \blacksquare \quad (2.7)$$

2.2.3. TSP euclidiano

Como se mencionó con anterioridad, hay un extenso estudio teórico sobre el TSP y en particular el TSP métrico. Un ejemplo de ello se puede ver en [2], en donde se propone un esquema de aproximación para la métrica euclidiana.

Si además se restringe el problema a un espacio métrico especifico, entonces podemos deducir nuevas propiedades. En este caso el espacio métrico será el plano euclidiano $\mathbb{R}^2$, para nuestros fines sólo ocuparemos el conjunto $\mathbb{Q}^2$. Sabiendo eso ahora para nuestra gráfica $G = (V, E)$ tenemos que el conjunto V son puntos en el plano euclidiano del tipo $(x, y) \in \mathbb{Q}^2$ y el conjunto de aristas serán todas las parejas de vértices, con la diferencia que el costo de moverme de un vértice i a un vértice j está dado por la distancia euclidiana entre estos dos vértices y será denotada por $d(i, j)$ que además cuenta con todas las propiedades de una métrica. Además de la representación también se sabe que TSP euclidiano (EucTSP por sus siglas en inglés) es NP-Duro [14]. La utilidad principal EucTSP está en la manera en que se representan los sitios a visitar, puesto que ahora al

ser un plano euclidiano sobre el que se está trabajando, todos los objetos son fácilmente representables a través de puntos en el plano.

Los dos algoritmos anteriores funcionan de la misma manera para EucTSP. La diferencia principal es la función de costo y la manera de representar los vértices. Existen diversos estudios sobre este problema y se pueden consultar[14], ya que de este problema nos interesa únicamente la representación.

2.3. Problema del camino más largo en gráficas dirigidas acíclicas

Un problema del que ya se habló en el capítulo previo, mejor conocido como problema del camino más corto o (SPP por sus siglas en inglés), es uno de esos problemas que surgen de la necesidad de reducir algún costo, que se puede representar como el traslado entre un par de vértices en una gráfica. De este modo parece no tener sentido hablar de encontrar el camino más largo en términos de costo, sin embargo, este problema nos puede ayudar en cierta forma a encontrar un camino que aproxime por ejemplo a un recorrido del TSP. Este problema a simple vista podría parecer que está en P como el SPP sin embargo esta suposición es falsa, puesto que se sabe que encontrar el camino áas largo en una gráfica (LPP por sus siglas en inglés) es NP-Duro [15].

Si agregamos una restricción a este problema, esta vez sobre la gráfica, que será un DAG, es posible resolverlo de manera exacta en tiempo polinomial [15], esta variante del LPP nos será de mucha utilidad en los problemas que se presentarán en las secciones siguientes y en los problemas tratados en este trabajo. Formalmente el problema sería. Dado un DAG $D = (V, A)$ con V el conjunto de vértices y A el conjunto de arcos una función $c : A \to \mathbb{Q}$ y un par de vértices u y v. Se busca encontrar el camino de costo máximo entre u y v.

Algoritmo 2.3 Algoritmo para caminos mas largos en un DAG

Entrada: Gráfica dirigida acíclica $D = (V, A)$ con $c : A \to \mathbb{Q}^+$
Salida: Camino mas largo en D

1: Ordenamiento-Topológico($D = (V, A)$)
2: **para cada** $v \in V$ en el orden topológico **hacer**
3: $\quad dist(v) = max_{(u,v)\in A}\{dist(u) + c(u, v)\}$
4: **fin para**
5: **devolver** $max_{v\in V}\{dist(v)\}$

2.4. El problema del reparador

Un problema relacionado es conocido coloquialmente como el problema del reparador o Traveling repairman problem (TRP por sus siglas en inglés). Este problema podría considerarse una versión más apegada a la realidad de lo que podría llegar a ser el TSP, debido a que éste último asume que las visitas del agente a sus destinos son instantáneas, que tiene un tiempo indefinido para hacer sus viajes así como para ser atendidos por los clientes y regresar a su lugar de origen. En contra parte el TRP habla de un agente que tiene igualmente que visitar sus destinos, pero además del costo de trasladarse de un lugar a otro, debe tomar en cuenta que ya estando en el lugar va a invertir una cantidad de tiempo en el destino, en el caso del TRP ese tiempo será usado en reparar algún desperfecto en el destino. Si continuáramos en el contexto del TSP podríamos suponer que ese tiempo hubiera sido utilizado en convencer de una venta a su cliente. Cambiando de nuevo al reparador también es plausible que los clientes que necesitan reparaciones no pueden atender al reparador a cualquier hora del día, así que los clientes (destinos) fijan un horario de atención, mejor conocido como ventana de tiempo.

En resumen en el TRP será necesario considerar tanto el costo (tiempo) de traslado, el tiempo necesario para realizar una tarea y las ventanas de tiempo en los destinos, con esta definición también podría presentarse el caso que el reparador realice más de una tarea en un destino y además de que el objetivo va a cambiar; en primera instancia ahora en vez de minimizar el tiempo total, podríamos pensar en minimizar el tiempo de espera entre cada tarea o bien maximizar el número de tareas que se pudiesen realizar, lo que tiene a su vez como consecuencia el ignorar el tiempo necesario para regresar desde la localización de la ultima tarea que se pudo realizar, al punto de origen.

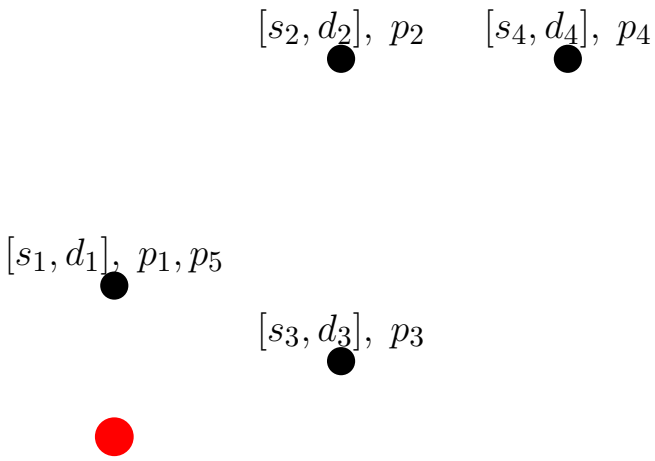

Figura 2.3: Ejemplo del problema del reparador (TRP) en la imagen se muestran un agente de color rojo y una serie de puntos negros en ciertas posiciones en el plano, una ventana de tiempo definida por $[s_i, d_i]$ con $d_i \geq s_i$ y una o varias tareas asociadas p_j.

Note que para que el reparador pueda hacer una tarea en algún sitio se debe cumplir que $p_j \leq d_i - s_i$ para $j \in i$ o bien si se desea que el reparador pueda hacer todas las tareas en un sitio i se debe cumplir que $\sum_{j \in i} p_j \leq d_i - s_i$.

Se sabe que TRP es NP-Completo aun cuando $p_j = 0$ para toda j [16].

2.5. TRP en una dimensión

Ahora tanto el reparador como los puntos a visitar, están sobre una línea recta. Este problema se le conoce como TRP en una dimensión ó TRP en una línea (Line-TRP por sus siglas en inglés), esto simplifica el espacio métrico del problema, sin embargo, no lo trivializa puesto que se sabe que es un problema NP-Completo aun cuando las tareas se realicen en tiempos instantáneos [16]. Existen diversas propuestas en cuanto a algoritmos de aproximación para este problema con tiempos de procesamiento instantáneos y para algunos casos especiales, pero de ello hablaremos en el siguiente capítulo.

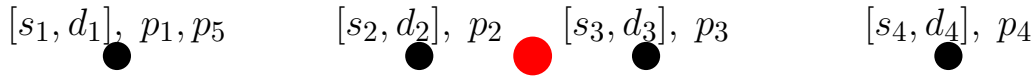

Figura 2.4: Variante de Line-TRP, ahora todos los sitios a visitar están sobre una línea.

Capítulo 3

Problema del agente viajero con objetivos móviles y sus variantes

Un problema que pudiera presentarse en la industria es que un posible destino a ser visitado no este fijo en un lugar determinado. Por ejemplo, hay barcos cisterna que abastecen de combustible a algunas embarcaciones, con el inconveniente de que las embarcaciones objetivo se siguen moviendo por el solo oleaje, así que el barco cisterna tiene que interceptar a la embarcación que necesita el combustible. Si además dicho barco tiene que atender a varias embarcaciones en una sola zarpada y regresar a su lugar de origen, entonces estamos hablando del problema del agente viajero en donde sus destinos (objetivos) son móviles. Algo similar sucede en la película *Iron Man 3*^{TM}. Cuando una serie de personas se precipitan en caída libre de un avión; *Iron Man*^{TM} tiene que atraparlos a todos antes de llegar al piso para salvarles la vida. En la película, el asistente virtual *Jarvis*^{TM} ejecuta un algoritmo ficticio que le crea la ruta para atraparlos a todos.

En vista de estas situaciones, ahora definiremos el problema del agente viajero con objetivos móviles (MTTSP por sus siglas en inglés) prepuesto en [10]. Se tiene un conjunto de n objetos móviles sobre un plano euclidiano, cada uno con una velocidad constante v_i y una posición inicial (x_i, y_i); además de un agente perseguidor con rapidez máxima U y con una posición inicial (x_0, y_0). Se quiere decidir si el agente puede capturar a los n objetos.

Problema: MTTSP
Entrada: Un conjunto de n objetos móviles cada uno con una posición inicial $p_i = (x_i, y_i) \in \mathbb{Q}^2$ además de una velocidad $v_i \in \mathbb{Q}^2$ y un agente con posición inicial $P_0 = (x_0, y_0)$ y rapidez máxima $U \in \mathbb{Q}$.
Salida: 1 si el agente puede capturar a todos los objetivos, 0 si no.

Se puede ver que este problema es una generalización del EucTSP puesto que en tal problema los objetivos tienen velocidad 0, con ello podemos que MTTSP es un problema NP-Completo.

3.1. MTTSP con pocos objetivos móviles

Una de las variantes estudiadas en [10], es aquella donde se restringe el número de objetos que se pueden mover con respecto al total. Una vez que sabemos que MTTSP es NP-Completo, ahora necesitamos de una medida que lo relacione con EucTSP, para ello se enuncia el siguiente resultado.

Teorema 3.1 (Helving et. al., 2003) *En una instancia del MTTSP con a lo más* $O\left(\frac{\log\ n}{\log\ \log n}\right)$ *objetivos móviles donde n es el total de objetivos (móviles + estáticos), puede ser aproximada en tiempo polinomial con una garantía de* $1+\alpha$*, donde* α *es una garantía de aproximación para una heurística arbitraria de TSP.*

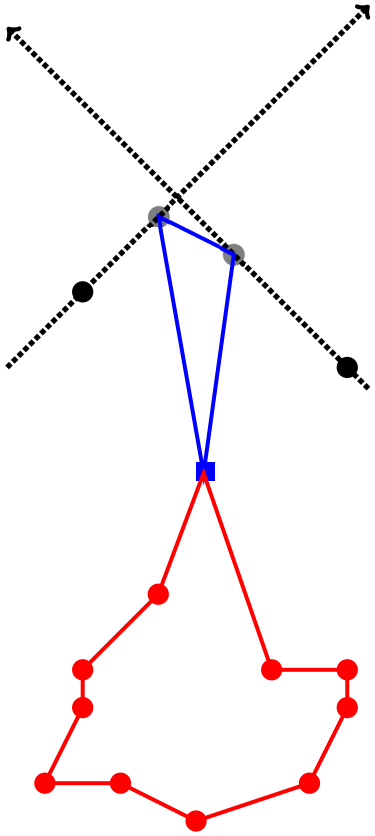

Figura 3.1: Ejemplo donde los objetos rojos son fijos y los objetos móviles son negros así como la ruta a seguir por el agente (cuadrado azul), desde su posición inicial.

3.2. MTTSP sobre una recta

Otra variante presentada en [10], es una donde todos los objetivos móviles y el agente se encuentran en una línea recta y se mueven sobre ella. Esto permite que se pueda implementar un algoritmo exacto de tiempo cuadrático [10].
Problema: MTTSP sobre una recta
Entrada: conjunto de n objetos móviles cada uno con una posición inicial $x_i \in \mathbb{Q}$ y una velocidad $v_i \in \mathbb{Q}$ y un agente que parte del origen con una rapidez máxima $U \in \mathbb{Q}$.
Salida: 1 si el agente puede capturar a todos los objetivos móviles, 0 si no.

El algoritmo de orden cuadrático se basa en dos principales resultados.

Lema 3.1 (Lema de no espera) *En cualquier recorrido óptimo de MTTSP, el agente siempre se mueve a máxima velocidad.*

Lema 3.2 *En un recorrido óptimo en el MTTSP sobre una recta, el agente no puede cambiar de dirección hasta que atrapa al objetivo más rápido y cercano a él.*

A partir de los resultados mencionados, se construye un algoritmo de programación dinámica con una serie de estados $A = (s_k, s_f)$, donde s_k representa el estado en el que un objetivo acaba de ser atrapado y s_f representa el siguiente objetivo móvil más rápido y cercano al agente, además de un estado inicial A_0 y un estado final A_{final}. Cada estado tiene asociada una función de tiempo $t(A)$ que representa el tiempo más corto para alcanzar el estado A, el estado inicial tiene asignado $t(A_0) = 0$. Note que el Lema 3.2 implica que sólo hay dos posibles transiciones para cada estado A. Estas transiciones representan a los dos siguientes objetivos que pueden ser capturados. Cada transición se representa con τ y el tiempo que se lleva en efectuar dicha transición es $t(\tau)$, que a su vez representa el tiempo necesario para capturar al objetivo correspondiente (ya sea el más rápido de la derecha o el de la izquierda), desde la posición s_k en el tiempo $t(A)$. Antes de construir la gráfica, el algoritmo hace un pre-procesamiento con los objetivos a la izquierda y a la derecha del agente, en donde los objetivos son ordenados de manera decreciente con respecto a sus velocidades y se borra de cada lista a los objetivos más cercanos al origen que su predecesor o dicho de otro modo, si es más lento pero está más cerca del origen, se quita. Puesto que el agente va a capturarlo cuando capture al más rápido sin cambiar de dirección, por lo que no es relevante para el resto del algoritmo.

Una vez hecho el pre-procesamiento, se tendrá un DAG $G_D = (V, A)$, donde V corresponde con los estados y A con las transiciones, además la parte acíclica se debe a que no es posible regresar y volver a capturar al mismo objetivo. Cada trayectoria en G_D se asocia con un camino desde el estado inicial al estado final. El algoritmo necesario no es más que una versión modificada de un algoritmo de caminos más cortos en un DAG, en donde el peso de los arcos depende del peso del estado actual hasta antes de la transición

l_j		Problema 1			Problema 2		Problema 3
		$k = 1$	$k = 2$	$k \geq 3$	$k = 1$	$k \geq 2$	
E	Un robot	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n^3)$	$O(n \log n)$
	Múltiples	$O(n \log n)$	$O(n^2)$	NP-C	$O(n \log n)$	NP-D	NP-D
S	Un robot	$O(n^2 \log n)$	$O(n^3 \log n)$	$O(n^3 \log n)$	$O(n^3 \log n)$	$O(n^5)$	$O(n^3 \log n)$
	Múltiples	$O(n^2 \log n)$	$O(n^6)$	NP-C	$O(n^3 \log n)$	NP-D	NP-D

Cuadro 3.1: Tabla extraída de [3]. *E* denota que los rayos donde están los agentes son parte de la entrada, *S* que son una variable de decisión del problema, NP-C es NP completo y NP-D es NP duro.

τ. El algoritmo se ejecuta en tiempo $O(|V| + |A|)$. Donde se necesitan a lo más n estados o vértices y a lo más n^2 transiciones.

3.3. Problemas de captura de pelotas

En [3] se propone otra variante de MTTSP, en donde los objetivos móviles se encuentran en el plano euclidiano y el o los agentes se encuentran sobre rayos que parten del origen. Algunos objetivos eventualmente se intersectarán con alguno de los rayos (los que nunca se intersectan simplemente son ignorados) en ese momento los agentes pueden capturarlos. A este problema se le conoce como Problema de captura de pelotas (BCP por sus siglas en inglés). Una propiedad adicional a este problema, es que el agente no tiene que regresar a su punto de partida, en este caso el origen.

El problema se define como sigue: Dado un conjunto $B = \{b_1, b_2 \dots b_n\}$ pelotas cada una con una posición inicial $p_i^0 = (x_i^0, y_i^0) \in \mathbb{Q}^2$ y velocidad $v_i = (v_i cos(\phi_i), v_i sen(\phi_i))$ y un conjunto $R = \{r_1, r_2 \dots r_k\}$ de robots homogéneos, cada uno con posición inicial en el origen y con velocidad máxima U, que se mueven sobre algún rayo l_j, como se muestra en la Figura 3.2. Resolviendo los siguientes tres problemas:

1. Decidir si k robots (agentes) pueden capturar n pelotas y calcular el itinerario de los robots.

2. Calcular el itinerario de k robots que maximice el número de pelotas capturadas.

3. Calcular un itinerario que minimice la cantidad de robots necesarios para capturar n pelotas.

En [3] los autores resumen los resultados de estos tres problemas, tomando en cuenta el numero de robots utilizados y si los rayos son parte de la entrada o son una variable de decisión. Un resumen de los resultados se puede ver en el Cuadro 3.1.

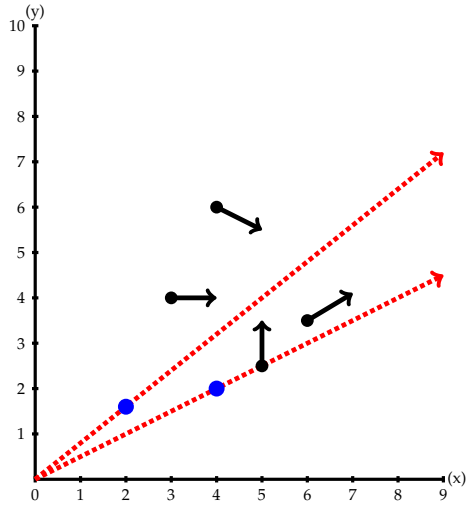

Figura 3.2: Ejemplo del problema de captura de pelotas con dos rayos, cada uno con un agente (azul), los objetivos móviles (negros) y como se intersectan con los rayos.

Cabe destacar que en el MTTSP sobre una recta los objetivos tienen una posición inicial, una velocidad y existen mientras no sean capturados. En el caso de los problemas tipo BCP los objetos existen sobre dicha recta solo por un instante de tiempo y en una posición de la recta. Por lo tanto es necesario determinar si el agente va a poder alcanzar la posición de aparición del objeto móvil en la recta. Si sólo fuese un agente sobre una recta, asumiéramos que todos los objetos llegarán en algún momento a dicha recta y la velocidad del agente fuera suficiente, entonces el orden en que se debe capturar a los objetos (pelotas) es el orden en que aparecen y el problema se reduce a encontrar ese orden, como se muestra en la figura 3.3.

El anterior problema es importante, ya que en nuestro caso se trabajará con una generalización de este, que al mismo tiempo es una generalización del TRP. La otra serie de resultados aunque son importantes dentro de los problemas BCP, no serán abordados con más profundidad que la mostrada en el Cuadro 3.1, ya que no son necesarios para este trabajo. Si el lector desea profundizar más, puede consultarlo en [3].

3.4. Algoritmos para Line TRP

Tanto en [4] como en [8] se proponen algoritmos para resolver Line TRP. En el caso de [4] se propone inicialmente una metodología basada en una versión más restringida de Line TRP, en donde las ventanas de tiempo son unitarias, dicha estrategia les permite diseñar un algoritmo con factor de aproximación 8 y después ampliar la estrategia a ventanas de tiempo generales.

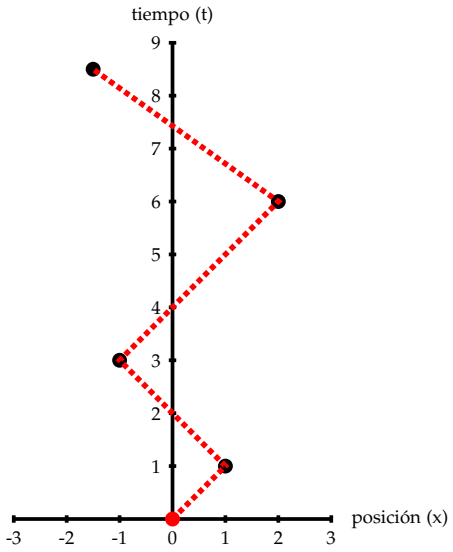

Figura 3.3: Diagrama Posición vs Tiempo donde se muestran objetos que aparecen sobre una recta únicamente un instante de tiempo y una ruta que los captura en el orden de aparición.

3.4.1. Algoritmos para Line TRP con ventanas unitarias

Bajo la suposición de que las tareas requieren un tiempo 0 para realizarse y además que todas tienen la misma ventana de tiempo. En un diagrama posición vs tiempo dichos objetos son representados mediante segmentos de recta de longitud unitaria y con pendiente 0. El objetivo en esta representación es encontrar una trayectoria limitada por la velocidad del agente (unitaria) que logre intersectar a la mayor cantidad posible de segmentos.

Para mayor facilidad primero le daremos una rotación de 45 grados al plano, de esa manera todas las trayectorias limitadas al cuadrante inducido por los vectores de velocidad del reparador.

Posteriormente utilizaremos una discretización del plano, en donde se divide empezando desde el origen, en filas de ancho k y en columnas del mismo tamaño, con $k = \frac{l}{\sqrt{2}}$, donde l es la longitud de los segmentos. Permitiendo mover ligeramente la rejilla de tal manera que los extremos de los segmentos no estén sobre la rejilla ni sobre sus intersecciones, como se muestra en la Figura 3.6. Este mecanismo tiene un inconveniente, ya que si un par de segmentos tienen coordenadas que los ponen lejos uno del otro, sera necesario trazar toda la rejilla aunque en medio no exista ningún segmento. Puesto que en el espacio vacío de la rejilla no se intersecta a ningún segmento, si se quita esa región del plano y se contrae la rejilla, la solución no se ve afectada, siempre y cuando no se traslapen los segmentos, como se muestra en la Figura 3.7. Esta corrección nos deja una rejilla de $2n \times 2n$.

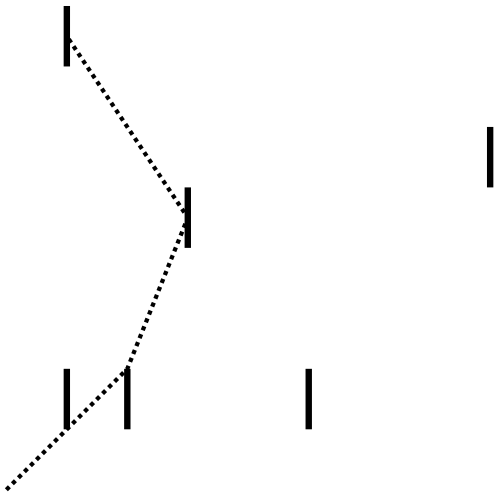

Figura 3.4: Ejemplo de Line TRP en el diagrama posición vs tiempo, se muestran distintos destinos de tiempo unitario y una posible ruta del reparador que maximiza la cantidad de destinos visitados.

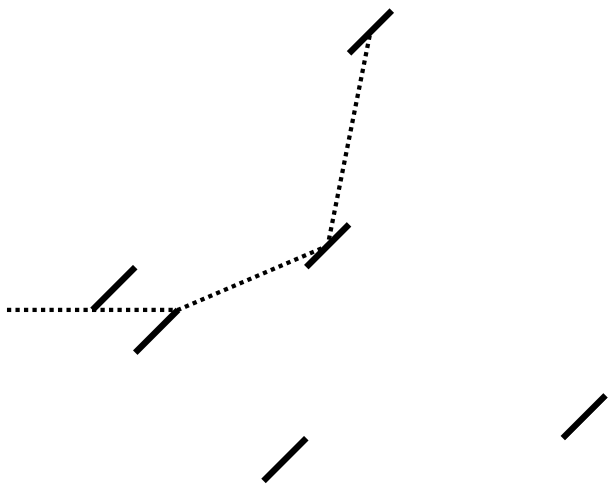

Figura 3.5: Diagrama con el plano rotado 45 grados a la derecha.

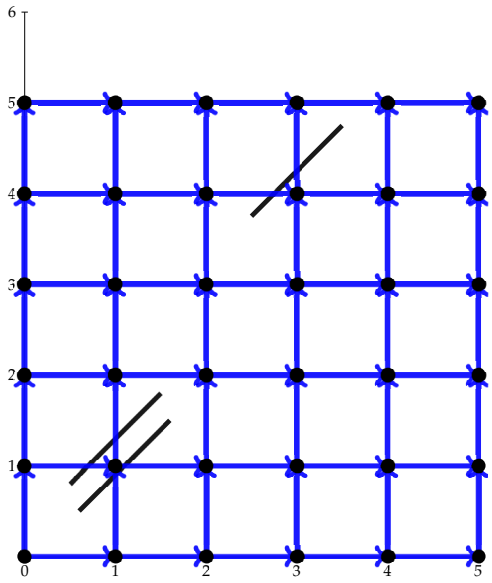

Figura 3.6: Ejemplo con tres segmentos de longitud $l = \sqrt{2}$, y la rejilla respectiva de tamaño unitario $k = 1$.

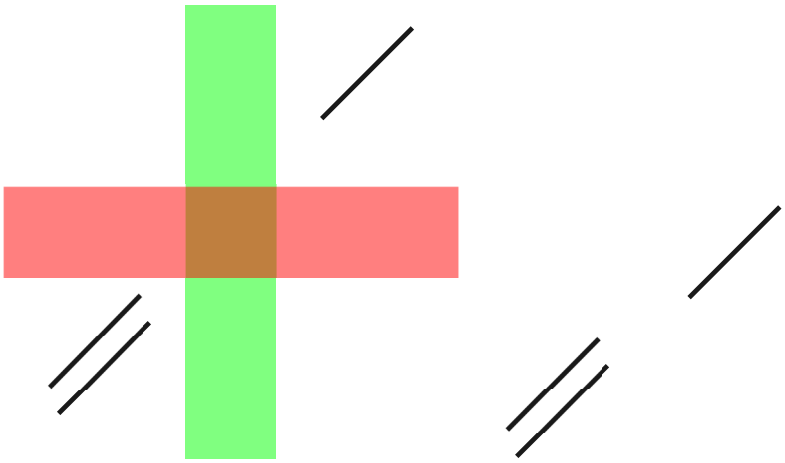

Figura 3.7: Ejemplo de corrección de la entrada. En la izquierda la región verde es la región vertical eliminada y la región roja es la región horizontal eliminada. A la derecha se muestra la nueva entrada después de eliminar y contraer las regiones verde y roja.

Una vez que se tiene la entrada corregida y la rejilla trazada sobre esta nueva entrada, se construirá una gráfica auxiliar $G_r = (V_r, A_r)$, donde los vértices (V_r) son las intersecciones de la rejilla y los arcos (A_r) son los segmentos entre cada par de intersecciones con dirección izquierda a derecha para vértices adyacentes horizontalmente y abajo hacia arriba para vértices adyacentes verticalmente (véase Figura 3.8).

Algoritmo 3.1 Algoritmo de aproximación con factor 8 para Line TRP con ventanas de tiempo unitarias

Entrada: Conjunto de n segmentos de longitud unitaria, rejilla unitaria
Salida: Camino de costo máximo en G_r

1: Construir la gráfica dirigida $G_r = (V_r, A_r)$.
2: A cada arco de A_r se le asigna como peso el número de segmentos que intersecta.
3: Calcular caminos más largos sobre G_r.
4: **devolver** Camino de costo máximo de G_r.

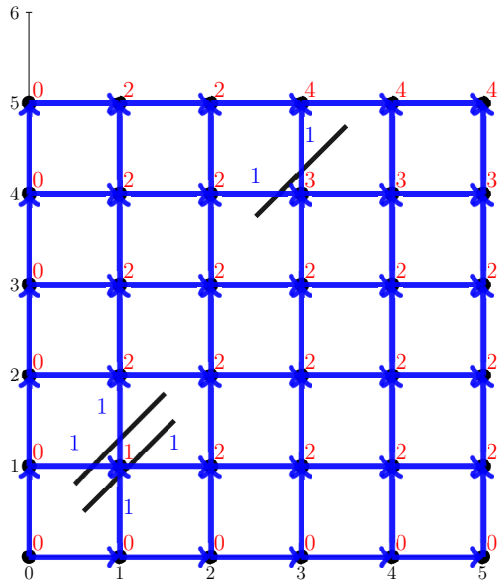

Figura 3.8: Ejemplo de ejecución del algoritmo 3.1, los números rojos representan el número máximo de objetos atrapados hasta ese punto

Como se ve en la Figura 3.8, el algoritmo 3.1 tiene almacenado en cada vértice el máximo número de segmentos intersectados hasta ese momento, sin embargo el conteo tiene un error ya que en el punto $(5,5)$ se tiene almacenado un 4 cuando sólo hay 3 segmentos en la rejilla. Por otro lado si seguimos la ruta de la Figura 3.9, dicha ruta sólo pasa por 2 segmentos pero se contabilizaron 4, esto debido al cambio de dirección en una sección de la rejilla que toca dos veces al mismo segmento.

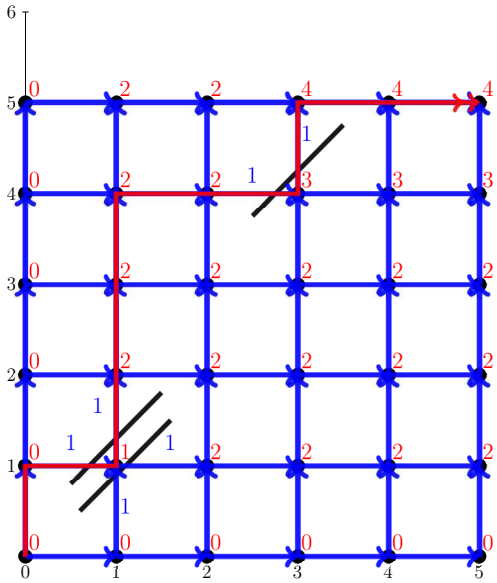

Figura 3.9: Error de doble conteo en los 2 segmentos que toca la ruta roja

Además de ese doble conteo, nada nos garantiza que una ruta óptima en la cuadrícula intersecte al mismo número de segmentos intersectados que una ruta óptima a través del plano. Ambas situaciones pueden resumirse en el Teorema 3.2.

Teorema 3.2 (Bar-Yehuda, 2005) *El factor de aproximación del algoritmo 3.1 es 8*

Demostración. La demostración del Teorema 3.2 requiere de los 2 siguientes lemas.

Lema 3.3 (Bar-Yehuda, 2005) *Sea q un camino y $k(q)$ el número de segmentos que intersectan a q. Sea $p^\star$ el camino óptimo sobre el plano y p' un camino óptimo restringido a la cuadrícula. Sea $k^\star = k(p^\star)$, $k' = k(p')$ y $k = k(p)$ donde p es el camino encontrado por el algoritmo 3.1 entonces $k \geq k'/2$.*

Lema 3.4 (Bar-Yehuda, 2005) *Sea q un camino y $k(q)$ el número de segmentos que intersectan a q. Sea $p^\star$ el camino óptimo sobre el plano y p' un camino óptimo restringido a la cuadrícula. Sea $k^\star = k(p^\star)$, $k' = k(p')$ y $k = k(p)$ donde p es el camino encontrado por el algoritmo 3.1 entonces $k' \geq k^\star/4$.*

El Lema 3.4 fue demostrado en [4] pero por completés se demostrará a continuación. *Demostración.* Supongamos que tenemos una ruta $p^\star$, dicha ruta es posible enmarcarla en bloques horizontales y verticales cuyo contorno coincide con la cuadrícula, como se muestra en la Figura 3.10. Se puede ver que, o bien se tienen al menos la mitad de bloques verticales del total de bloques ó bien, se tienen al menos la mitad de bloques horizontales del total de bloques. En este caso supondremos que los bloques horizontales cumplen

esa propiedad, aunque la demostración es análoga para el caso de los bloques verticales. Ahora, construyamos un camino p_r a través de los bloques. Para un bloque horizontal tenemos únicamente dos posibilidades:

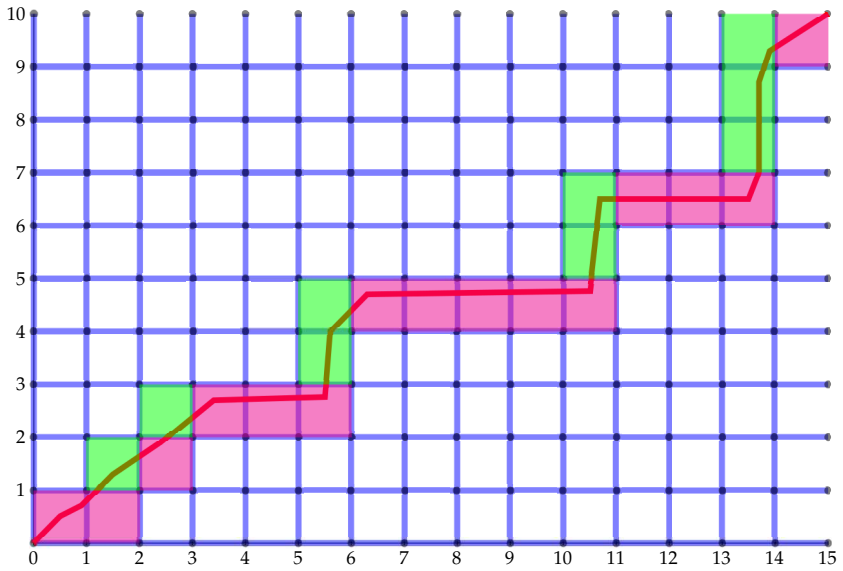

Figura 3.10: Descomposición de una trayectoria óptima ($p^\star$) en bloques horizontales (rosas) y verticales (verdes).

1. El camino empieza en la esquina inferior izquierda con dirección izquierda derecha y después abajo a arriba hasta la esquina superior derecha del bloque.
2. El camino empieza por la esquina inferior izquierda con dirección abajo arriba y después izquierda derecha hasta la esquina superior del bloque.

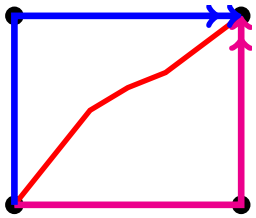

Figura 3.11: Caminos posibles sobre un bloque, de rosa el (1) y de azul el (2).

Un recorrido por la rejilla, a través de un bloque horizontal requiere que se seleccione uno de los dos caminos anteriormente mencionados. Se elige entonces el camino que intersecte a la mayor cantidad de segmentos. Adicionalmente se sabe que los segmentos solo pueden intersectar al camino (1) o al camino (2) pero no a ambos, de no ser así

la longitud de los segmentos seria mas grande que la establecida. Se puede ver que el camino elegido intersecta al menos a la mitad de segmentos que intersectaba el bloque horizontal respectivo, podemos decir que los caminos por los bloques horizontales son una 2-aproximación. Por tanto la aproximación de p_r es $1/4p^\star$. ■

Teniendo esto es fácil deducir los valores. $2k \geq k'$ entonces $2k \geq k^\star/4$ y por lo tanto $k \geq k^\star/8$. ■

El Lema 3.4 se utilizara en un resultado que será expuesto en capítulos posteriores, puesto que en el se hace uso de la misma cuadrícula unitaria.

Además del algoritmo descrito anteriormente, existen 2 propuestas adicionales, la primera es una mejora en cuanto al factor de aproximación $(4 + \epsilon)$ pero a un costo computacional muy alto ($O(n^{8/\epsilon})$) esta propuesta también es presentada en [4], por otro lado en [8] se propone un algoritmo de factor 3 pero con un costo computacional de $O(n^4)$.

3.4.2. Algoritmos para Line-TRP con ventanas de tiempo generales

En el caso de ventanas de tiempo arbitrarias, en [4] se propone dividir el problema dependiendo de las longitudes de las ventanas de tiempo, creando un subconjunto para los segmentos que están entre $[2^i, 2^{i+1})$ con $i = 0, 1, \ldots p$, posteriormente se resuelve cada subconjunto de segmentos por separado con el algoritmo 3.1 y se toma el que tenga un valor más alto. Al final emplear este mecanismo llevará el factor de aproximación hasta 16. Por otro lado, en la misma referencia se prepone otro algoritmo basado en unas estructuras de datos llamadas combs, el uso de estas estructuras permite reducir el factor de aproximación a 12.

Capítulo 4
Captura de objetos móviles sobre una recta con programación lineal

En el capítulo anterior se habló de dos variantes de MTTSP. En una de ellas, la de [10], los puntos móviles a capturar están sobre una misma línea recta, se comienza en el origen y no se tiene que regresar a ese mismo punto. Para esa variante se muestra un algoritmo de orden cuadrático que lo resuelve. Es a partir de esta variante como llegamos a los problemas que plantea este trabajo y en particular este capítulo, en donde la diferencia principal radica en que los objetos tienen un intervalo de existencia sobre la recta y una posición de aparición, como se muestra en la Figura 4.1.

El primer problema, que llamaremos captura de todos los objetos móviles sobre una recta (CTOMSR), se define como sigue: Dado un agente con rapidez variable (acotada superiormente por una constante U) y n objetos móviles, cada uno con un momento de aparición (a_i), un momento de desaparición (d_i), una posición de aparición (p_i) y una velocidad (v_i), se quiere decidir si el agente es capaz de capturar todos los objetos en el orden $1, 2, \ldots, n$. En caso de que sí se pueda, se pueden formular otros dos problemas: por un lado se desea minimizar el tiempo de captura (CTOMSR rápido) y por otro lado se desea minimizar la distancia total recorrida (CTOMSR perezoso).

En estos problemas dar un orden predefinido de captura tiene sentido, porque en principio se tiene que determinar si es posible capturar a todos los objetos, situación que marca una diferencia respecto al Euclidean TSP donde ello resulta ser trivial. Se sabe que el problema de determinar si es posible capturar a todos los objetos sin un orden predefinido de captura es NP Completo [16]. Por otro lado, si además no se tuvieran intervalos de tiempo se sabe que el problema se puede resolver en tiempo polinomial [10]. Esta situación intermedia determina la importancia de los tres problemas a tratar.

4.1. CTOMSR

Ahora podemos ubicar la posición y el tiempo en un plano, con lo cual los puntos móviles sobre la recta se convertirán en segmentos de recta en el plano posición vs tiempo

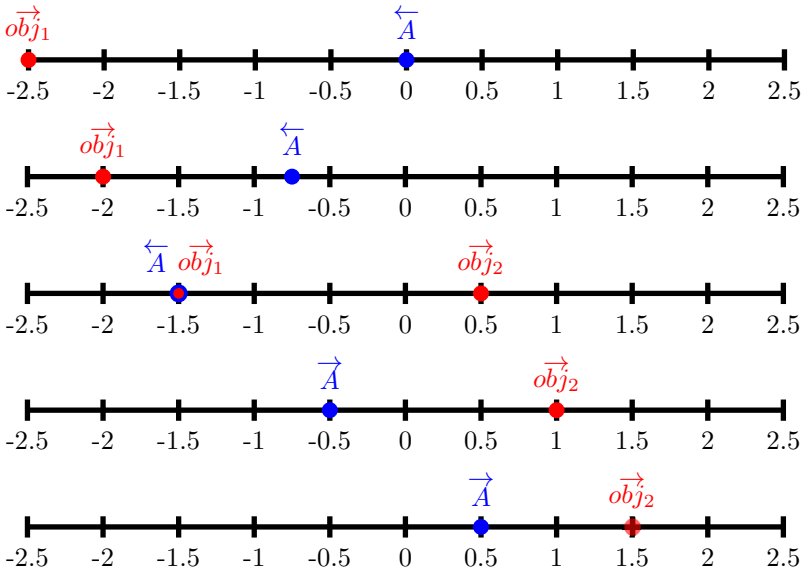

Figura 4.1: Ejemplo del problema propuesto que muestra cómo se mueve el agente con rapidez entre 0 y 1 en la recta, a la izquierda con rapidez 0.75 y a la derecha con rapidez 1 (en distintos periodos de tiempo), así como la aparición y desaparición de objetos móviles.

(x vs t).

Si x_i y t_i son la posición y el momento de captura del objeto i, es necesario que se capture en el intervalo de existencia del objeto, es decir:

$$a_i \leq t_i \leq d_i.$$

También se necesita que dicho punto esté sobre la recta que describe el segmento que queremos intersectar, es decir:

$$x_i - p_i = v_i(t_i - a_i).$$

Además, el agente debe poder capturar al objeto o, dicho de otro modo, el objeto debe estar en el rango de intercepción del agente, es decir:

$$-U(t_i - t_{i-1}) \leq x_i - x_{i-1} \leq U(t_i - t_{i-1}).$$

Adicionalmente se necesita la relación de orden de captura, es decir:

$$t_{i-1} \leq t_i.$$

El modelo descrito se puede visualizar en la Figura 4.2.

Aunque la posición inicial del agente pudiera ubicarse en cualquier punto de la recta, sin pérdida de generalidad supondremos que el agente partirá del origen, es decir $t_0 = 0$

y $x_0 = 0$. Por último, sólo nos queda decir que el tiempo es no negativo $t_i \geq 0$ y la posición x_i es libre porque se puede capturar cualquier objeto en cualquier parte de la recta donde el objeto exista.

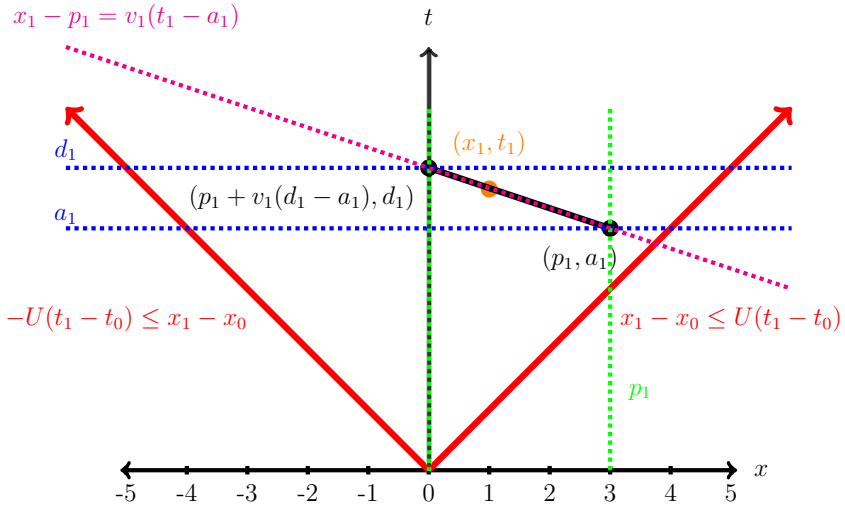

Figura 4.2: Diagrama x vs t con todas las restricciones del objeto 1.

El modelo lineal completo del problema de capturar todos los objetos móviles sobre una recta (CTOMSR) queda como sigue:

$$
\begin{aligned}
t_i & \geq a_i && i = 1, \ldots, n && (4.1)\\
t_i & \leq d_i && i = 1, \ldots, n && (4.2)\\
-v_i t_i + x_i & = p_i + -v_i a_i && i = 1, \ldots, n && (4.3)\\
Ut_i - Ut_{i-1} + x_i - x_{i-1} & \geq 0 && i = 1, \ldots, n && (4.4)\\
-Ut_i + Ut_{i-1} + x_i - x_{i-1} & \leq 0 && i = 1, \ldots, n && (4.5)\\
-t_i + t_{i-1} & \leq 0 && i = 1, \ldots, n && (4.6)\\
t_0 & = 0 && && (4.7)\\
x_0 & = 0 && && (4.8)\\
t_i & \geq 0 && i = 1, \ldots, n && (4.9)\\
x_i & \text{ libre} && i = 1, \ldots, n && (4.10)
\end{aligned}
$$

Observe que la restricción (4.6) se puede quitar del modelo puesto que queda implícita en la restricciones (4.4) y (4.5).

4.2. CTOMSR rápido

Para el caso del CTOMSR rápido, es posible despejar la variable x_i en la igualdad (4.3) del modelo anterior, el nuevo modelo quedaría en términos de t_i:

$$\begin{aligned}
\text{mín}\, z =& t_n && && (4.11)\\
t_i \geq& a_i && i = 1,\ldots,n && (4.12)\\
-t_i \geq& -d_i && i = 1,\ldots,n && (4.13)\\
(U - v_i)t_i - (U + v_{i-1})t_{i-1} \geq& \alpha_i && i = 1,\ldots,n && (4.14)\\
(U + v_i)t_i - (U - v_{i-1})t_{i-1} \geq& -\alpha_i && i = 1,\ldots,n && (4.15)\\
t_0 =& 0 && && (4.16)\\
t_i \geq& 0 && i = 1,\ldots,n && (4.17)
\end{aligned}$$

donde las $\alpha_i = a_i v_i - a_{i-1} v_{i-1} - p_i + p_{i-1}$ son constantes.

Con el programa lineal anterior y los siguientes resultados podemos decir que tanto el problema de la factibilidad como el de minimización de tiempo, se pueden resolver en tiempo polinomial partiendo del algoritmo de Karmarkar [12].

Teorema 4.1 (Karmarkar) *La complejidad del algoritmo de Karmarkar para un programa lineal de la forma* mín cx *sujeto a* $Ax \geq b$ *con* p *variables y* q *restricciones es de*

$$O(p^{3.5}(|A| + |b| + |c|)^2),$$

donde $|A|$, $|b|$ *y* $|c|$ *son la cantidad de bits necesarios para codificar la matriz* A, b *y* c *respectivamente.*

De donde se desprende el siguiente corolario que muestra el resultado particular para CTOMSR rápido.

Corolario 4.1 *Para el problema* CTOMSR rápido *existe un algoritmo basado en el algoritmo de Karmarkar y se ejecuta en tiempo*

$$O(\ell^2 n^{5.5})$$

donde ℓ *es la máxima cantidad de bits que se requieren para almacenar* U *y cualquier* a_i, d_i, p_i *o* v_i.

Demostración. Con el modelo de minimización del tiempo de captura (4.11–4.17) podemos obtener los siguientes tamaños:

$$|A| = (8\ell + 2)n, \quad |b| = 6\ell n, \quad |c| = 1.$$

El número de operaciones requeridas se puede obtener mediante el Teorema 4.1 y es $O(n^{3.5}(14\ell n + 2n + 1)^2)$, lo cual está en $O(\ell^2 n^{5.5})$. ■

Aun cuando se puede resolver en tiempo polinomial, está claro que mediante este algoritmo general se tiene un exponente muy grande, además dependeremos de la entrada misma y de su codificación.

4.2.1. Algoritmo de tiempo lineal para CTOMSR rápido

A continuación presentaremos un algoritmo que resuelve CTOMSR rápido en tiempo lineal. Para ello resolveremos el problema por etapas. Comenzando en el origen, se determina el primer (t_1^e) y el último momento (t_1^f) en que se puede capturar el primer objeto. Con esa información se calcula lo mismo para el segundo objeto y así sucesivamente hasta calcular t_n^e y t_n^f. Es decir, en cada etapa se resuelven dos programas lineales con funciones objetivo $t_i^e = \text{mín}(t_i)$ y $t_i^f = \text{máx}(t_i)$ y el conjunto de restricciones (4.18–4.24), para las variables t_i y t_{i-1}:

$$t_i \geq a_i \tag{4.18}$$
$$-t_i \geq -d_i \tag{4.19}$$
$$t_{i-1} \geq t_{i-1}^e \tag{4.20}$$
$$-t_{i-1} \geq -t_{i-1}^f \tag{4.21}$$
$$(U - v_i)t_i - (U + v_{i-1})t_{i-1} \geq \alpha_i \tag{4.22}$$
$$(U + v_i)t_i - (U - v_{i-1})t_{i-1} \geq -\alpha_i \tag{4.23}$$
$$t_i \geq 0 \tag{4.24}$$

donde $\alpha_i = a_i v_i - a_{i-1} v_{i-1} - p_i + p_{i-1}$ son constantes, además t_i^e y t_i^e se calculan en el paso anterior, con la excepción de $t_0^e = 0$ y $t_0^f = 0$. Estos programas lineales se deducen del programa lineal para CTOMSR rápido (4.11–4.17).

En este momento tenemos el tiempo mínimo y el tiempo máximo para capturar a todos los objetos. Si además se desea encontrar una ruta factible para dicho tiempo mínimo, se toma $t_n'^e = t_n^e$, $t_n'^f = t_n^e$ y se resuelven dos programas lineales con funciones objetivo $t_{i-1}'^e = \text{mín}(t_{i-1})$ y $t_{i-1}'^f = \text{máx}(t_{i-1})$ y el conjunto de restricciones (4.18–4.24), para las variables t_i y t_{i-1}, modificando las dos primeras restricciones como sigue:

$$t_i \geq t_i'^e \tag{4.25}$$
$$-t_i \geq -t_i'^f \tag{4.26}$$

donde $t_i'^e$ y $t_i'^f$ se calculan en el paso anterior.

Adicionalmente se buscará la ruta que ocupará el tiempo más tardío mediante un procedimiento similar pero con $t_n''^e = t_n^f$, $t_n''^f = t_n^f$ y el conjunto de restricciones (4.18–4.24), para las variables t_i y t_{i-1}, modificando las dos primeras restricciones como sigue:

$$t_i \geq t_i''^e \tag{4.27}$$
$$-t_i \geq -t_i''^f \tag{4.28}$$

donde $t_i''^e$ y $t_i''^f$ se calculan en el paso anterior.

Algoritmo 4.1 Algoritmo por etapas para CTOMSR rápido

Entrada: Conjunto de n objetos, con parámetros a, d, p, v y un agente con rapidez máxima U

Salida: Región factible para CTOMSR rápido.

1: Sea $t_0^e = 0$, $t_0^f = 0$.
2: **para** $i = 1$ **hasta** n **hacer**
3: Resolver los programas lineales $t_i^e = \text{mín}(t_i)$ sujeto a (4.18–4.24) y $t_i^f = \text{máx}(t_i)$ sujeto a (4.18–4.24).
4: **fin para**
5: Sea $t_n'^e = t_n^e$, $t_n'^f = t_n^e$.
6: **para** $i = n$ **hasta** 1 **hacer**
7: Resolver los programas lineales $t_{i-1}'^e = \text{mín}(t_{i-1})$ sujeto a (4.25–4.26) y (4.20–4.24) y $t_{i-1}'^f = \text{máx}(t_{i-1})$ sujeto a (4.25–4.26) y (4.20–4.24).
8: **fin para**
9: Sea $t_n''^e = t_n^f$, $t_n''^f = t_n^f$.
10: **para** $i = n$ **hasta** 1 **hacer**
11: Resolver los programas lineales $t_{i-1}''^e = \text{mín}(t_{i-1})$ sujeto a (4.27–4.28) y (4.20–4.24) y $t_{i-1}''^f = \text{máx}(t_{i-1})$ sujeto a (4.27–4.28) y (4.20–4.24).
12: **fin para**

Teorema 4.2 *El Algoritmo 4.1 resuelve el problema* CTOMSR rápido *en tiempo lineal.*

Demostración. Cada uno de los programas lineales que se deben resolver en el paso 3 del algoritmo consta de 2 variables y 6 desigualdades. Se sabe por la teoría de la programación lineal que la solución óptima se puede obtener resolviendo a lo más $\binom{2+6}{6} = 28$ sistemas de 6 ecuaciones lineales con 6 variables. Como cada uno de estos sistemas se puede resolver en tiempo constante, entonces el paso 3 se puede llevar a cabo también en tiempo constante. Es decir, el paso 2 del algoritmo se puede completar en tiempo lineal. Evidentemente, lo mismo es cierto para los pasos 6 y 10 del algoritmo. Finalmente, como los pasos 1, 5 y 9 son de tiempo constante, el tiempo de ejecución del algoritmo es lineal. ■

En la Figura 4.3 se muestra una salida del algoritmo por etapas. Los segmentos de línea continua son los objetos móviles. La zona de líneas verticales y contorno punteado corresponde con la región a través de la cual cualquier ruta requiere de tiempo mínimo t_n^e para capturar a todos los objetos. La zona de líneas horizontales y contorno rayado corresponde con la región a través de la cual cualquier ruta requiere de tiempo máximo t_n^f para capturar a todos los objetos. La región factible está determinada por la región de tiempo máximo, la región de tiempo mínimo y la región sin ningún patrón entre estas dos.

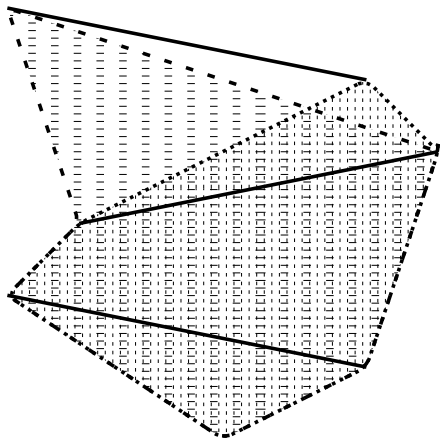

Figura 4.3: Ejemplo de la salida del Algoritmo 4.1.

4.3. CTOMSR perezoso

Otra métrica que se podría minimizar además del tiempo de captura, es la distancia que recorre el agente, el interés podría desprenderse de un ahorro en la energía que necesite emplear el agente para trasladarse y minimizar posiblemente un costo de operación.

Para el caso del CTOMSR perezoso, el modelo ahora requiere del despeje de las t_i y queda en términos de x_i (suponiendo sin pérdida de generalidad que ninguna $v_i = 0$).

$$\text{mín } z = \sum_{i=1}^{n} r_i \tag{4.29}$$

$$\frac{x_i}{v_i} \geq \frac{p_i}{v_i} \qquad i = 1, \ldots, n \tag{4.30}$$

$$-\frac{x_i}{v_i} \geq a_i - d_i - \frac{p_i}{v_i} \quad i = 1, \ldots, n \tag{4.31}$$

$$-x_i + x_{i-1} + r_i \geq 0 \qquad i = 1, \ldots, n \tag{4.32}$$

$$x_i - x_{i-1} + r_i \geq 0 \qquad i = 1, \ldots, n \tag{4.33}$$

$$\left(\frac{U}{v_i} + 1\right) x_i - \left(\frac{U}{v_{i-1}} + 1\right) x_{i-1} \geq \delta_i \qquad i = 1, \ldots, n \tag{4.34}$$

$$\left(\frac{U}{v_i} - 1\right) x_i - \left(\frac{U}{v_{i-1}} - 1\right) x_{i-1} \geq \delta_i \qquad i = 1, \ldots, n \tag{4.35}$$

$$x_0 = 0 \tag{4.36}$$

$$r_i \geq 0 \qquad i = 1, \ldots, n \tag{4.37}$$

$$x_i \quad \text{libre} \qquad i = 1, \ldots, n \tag{4.38}$$

en donde las r_i son variables auxiliares para eliminar un valor absoluto de la función objetivo que era originalmente $\sum_{i=1}^{n} |x_i - x_{i-1}|$ y convertirla en (4.29) y con

$$\delta_i = U \left(a_{i-1} - \frac{p_{i-1}}{v_{i-1}} - a_i + \frac{p_i}{v_i} \right)$$

constante.

Note que si alguna $v_i = 0$, entonces $x_i = p_i$ y el modelo se simplificaría.

Corolario 4.2 *Para el problema* CTOMSR perezoso *existe un algoritmo basado en el algoritmo de Karmarkar y se ejecuta en tiempo*

$$O(\ell^2 n^{5.5})$$

donde ℓ es la máxima cantidad de bits que se requieren para almacenar U y cualquier a_i, d_i, p_i o v_i.

Demostración. Con el modelo de minimización de la distancia total recorrida (4.29–4.38) podemos obtener los siguientes tamaños:

$$|A| = (10\ell + 6)n, \quad |b| = 8n\ell, \quad |c| = n.$$

El número de operaciones requeridas se puede obtener mediante el Teorema 4.1 y es $O(n^{3.5}(18n\ell + 7n)^2)$ lo cual está en $O(\ell^2 n^{5.5})$. ■

Capítulo 5

Captura de objetos móviles sobre una recta con programación dinámica

5.1. Capturar la máxima cantidad de objetos móviles

En el caso de que no sea posible capturar a todos los objetos móviles, surge la pregunta de cual fue el número máximo de objetos capturados, en este caso no se tomara en cuenta el orden de captura, a este problema lo llamaremos máx COMSR, para ello utilizaremos la siguiente estrategia de discretización de rutas.

Primero se crea una rejilla en el plano *posición vs tiempo* donde cada objeto móvil se representa con un segmento de recta. En cada extremo del segmento se traza una línea paralela a las rectas $t = Ux$ y a $t = -Ux$ donde t es el tiempo y x es la posición, como se muestra en la Figura 5.1. Con la excepción de los segmentos que quedan totalmente fuera del cono formado las rectas $-Ux$,Ux, que son ignorados. Por otro lado, si queda parcialmente fuera, es decir, una o dos porciones del segmento quedan fuera, se quita la porción(es) que quedó fuera y el punto de intersección con la recta $t = -Ux$ o la recta $t = Ux$ sería el nuevo punto extremo del segmento.

Con lo anterior definimos un nuevo sistema coordenado donde las rectas con pendiente negativa definen el primer elemento del par ordenado y las rectas con pendiente positiva forman el segundo elemento del par ordenado.

Observación 5.1 *Si existen dos rectas con la misma ordenada al origen sólo se toma en cuenta una. Así las rectas izquierdas (con pendiente negativa) sólo se intersectan con las rectas derechas (con pendiente negativa) y no dentro del mismo conjunto.*

Además de esta observación tenemos otro par de observaciones con respecto de la ubicación de los segmentos dentro de la cuadrícula formada.

Observación 5.2 *Ningún segmento de recta (objetos móviles) tienen un punto extremo en el interior de alguno de los paralelogramos de la cuadrícula.*

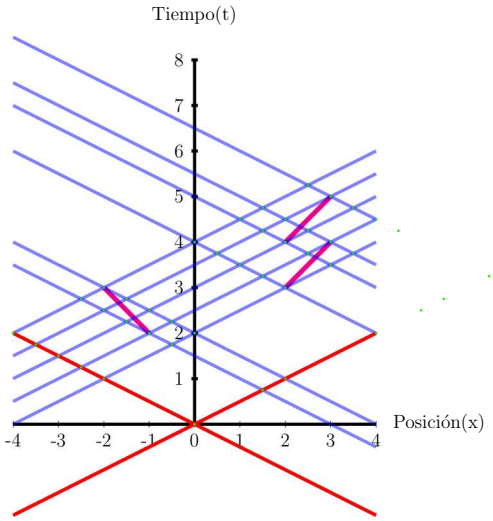

Figura 5.1: Ejemplo del procedimiento de discretización de las posibles rutas

Esta observación se puede verificar fácilmente, ya que si existiera un segmento que tenga un extremo dentro de unos de estos paralelogramos, por la construcción de la cuadrícula en ese punto extremo debería pasar un par de rectas con pendientes $-U$ y U, como se muestra en la Figura 5.2.

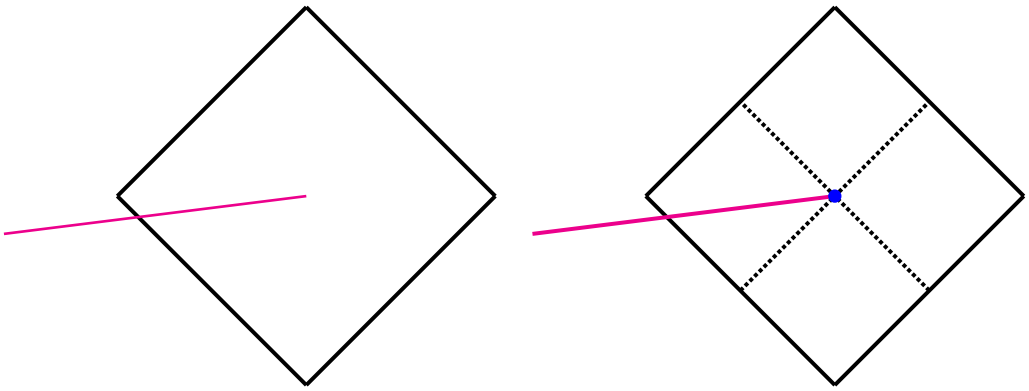

Figura 5.2: Ejemplo de la Observación 5.2

Observación 5.3 *Los extremos de los segmentos sólo se ubican en las intersecciones de las rectas que definen la cuadrícula.*

Un caso similar al anterior, ya que si uno de estos segmentos tuviesen uno de sus extremos en una part de las rectas de la cuadrícula, entonces ahí se debió definir un par de rectas con pendientes $-U$ y U, que a su vez definirían una intersección nueva. Por lo que sólo es posible que los extremos de los segmentos se ubiquen en las intersecciones de las rectas que definen la rejilla, como se observa en la Figura 5.3.

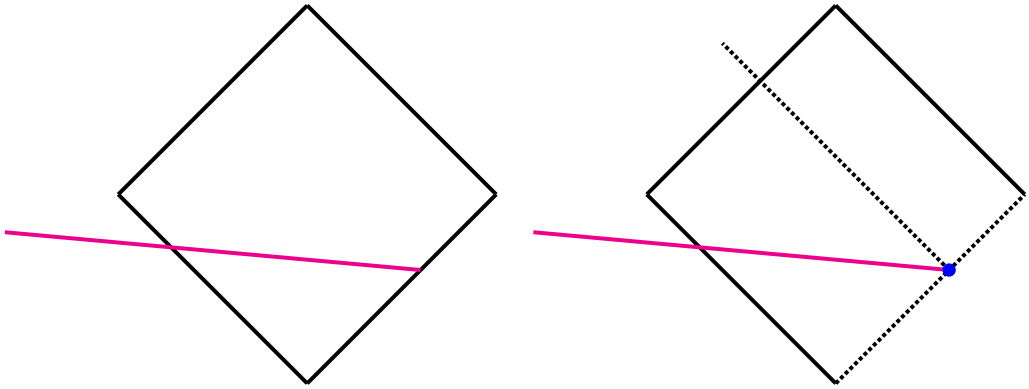

Figura 5.3: Ejemplo de la Observación 5.3

Con base en estas observaciones y con las definiciones previas ahora enunciaremos el siguiente resultado.

Teorema 5.1 *Sea p una ruta cualquiera con pendiente entre $-U$ y U desde el origen hasta algún objeto κ. Si dicha ruta intersecta a k segmentos entonces existe una ruta lineal en partes que intersecta al menos a los mismos k segmentos.*

Demostración. Dado que la ruta que intersecta a k segmentos existe, entonces para moverse de un segmento a otro se tiene una curva que describe el desplazamiento del agente entre ese par de segmentos, con la condición de que para cualquier tiempo t con $0 \leq t \leq t(k)$ $U(t) \leq U$, por el teorema de la velocidad media se sabe que una distancia recorrida en un tiempo determinado por un objeto puntual con aceleración, entonces, otro objeto con velocidad constante puede recorrer la misma distancia en el mismo tiempo cumpliendo la siguiente propiedad

$$U_{promedio} = \frac{x}{t} = \frac{1/2at^2}{t} = \frac{1}{2}at = \frac{1}{2}U_{final}$$

donde U_{final} es la velocidad que se tiene en el punto de intersección de la trayectoria del agente con los segmentos de recta, es así como entre un par de segmentos podemos remplazar una curva cualquiera por una recta con pendiente constante.

Posteriormente si elegimos otro par de objetos podemos seguir el mismo procedimiento esto, no puede garantizar que las pendientes entre cada par de objetos sean iguales, pero sí se puede garantizar que sean constantes por lo que la trayectoria describe un comportamiento piezolineal, además de ello la trayectoria original tocaba k segmentos, como en la nueva trayectoria entre cada par de objetos se remplaza la trayectoria pero se conservan los puntos inicial y final de dichas trayectorias y que corresponden a los puntos de intersección entonces la nueva trayectoria piezolineal intersecta los mismos k objetos. ■

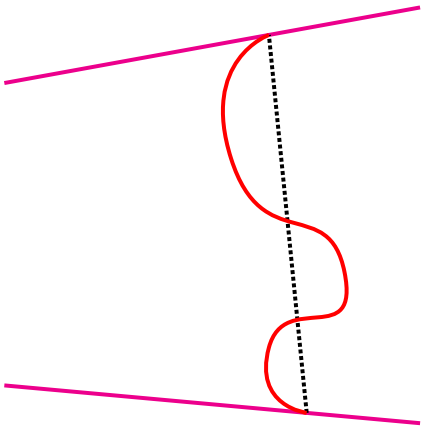

Figura 5.4: La ruta en línea punteada es la obtenida al aplicar el Teorema 5.1

Teorema 5.2 *Sea p una ruta lineal en partes con pendiente entre $-U$ y U desde el origen hasta algún objeto κ. Si dicha ruta intersecta a k segmentos entonces existe una trayectoria basada en la construcción de la rejilla con rapidez máxima que intersecta al menos a los mismos k segmentos.*

Demostración. En este caso debemos concentrarnos en lo que sucede al interior de cada paralelogramo inducido por la cuadrícula, ya que debido a que la trayectoria original es una trayectoria piezolineal, con una pendiente limitada tanto superior como inferiormente. Dicha ruta atraviesa una serie de paralelogramos de la rejilla y la forma en la que los atraviese se puede visualizar en la Figura 5.5.

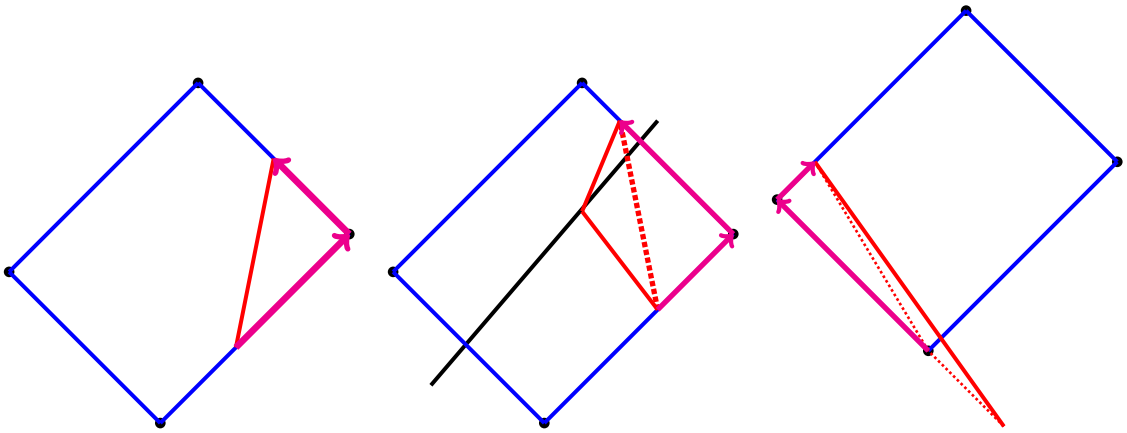

Figura 5.5: Casos que pueden presentarse paralelogramo una ruta original (roja) pasa por cada cuadro y su descomposición en una ruta por la rejilla (rosa): a) Si pasa formando un triangulo, b) Si cambia de dirección porque intersectó un objeto dentro del cuadro, c) Si la ruta forma un cuadrilátero más pequeño.

Note que es muy sencillo convertir una ruta por en medio de un cuadro a una que vaya por su perímetro y además es fácil ver que aunque hubiese un segmento dentro

debido a la observación 5.2, aun se pueden intersectar a los segmentos. Sin embargo, para el caso c se pueden presentar el subcaso de la Figura 5.6.

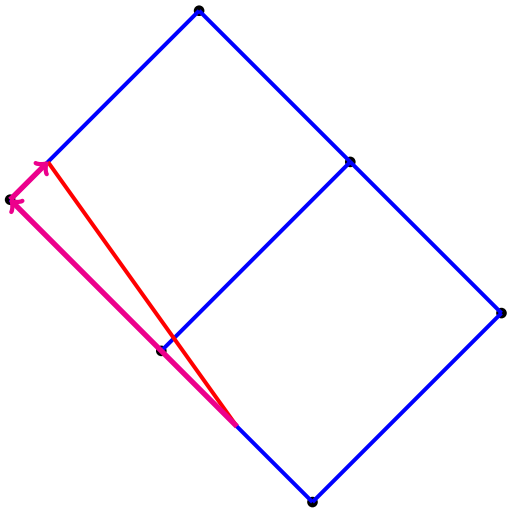

Figura 5.6: Caso en que la ruta no es paralela a la cuadrícula

En este caso es posible otra vez remplazarla con la otra parte del triángulo formado con la ruta p (roja) y la rejilla, sólo que en este caso el triángulo abarcará más de un paralelogramo.

Por otro lado, otra situación que podría suceder en la conversión de la ruta, es que eventualmente esa ruta tenga que dar vuelta y lo haga en forma paralela a la rejilla como en la Figura 5.7. Entonces la ruta original se puede cambiar por la ruta en la rejilla más próxima, esto no afecta a los objetos que intersecta la ruta puesto que entra y sale del conjunto de paralelogramos tan rápido como la ruta original, pero en momentos y posiciones anteriores que la ruta original.

Teniendo estos casos ahora es posible replicar este procedimiento desde el origen hasta el final de la ruta original. Es posible tener una ruta por la rejilla desde el inicio al final de la ruta p, además la ruta p captura a k objetos. La ruta por la rejilla intersecta a los mismos k objetos puesto que en cada paralelogramo o bien se preserva el inicio y fin de la ruta que pasa por ahí o bien la ruta preserva el inicio y final pero de un conjunto de paralelogramos o bien se cambian por un inicio y final equivalentes y además por las observaciones 5.2 y 5.3 se puede aun seguir capturando a los objetos sobre la rejilla. ■

5.2. Algoritmo de aproximación de factor 4 para ventanas unitarias y $v = 0$

En esta sección se trabajará sobre una versión restringida del problema donde los objetos carecen de movimiento y además su periodo de existencia sobre la recta es unitario.

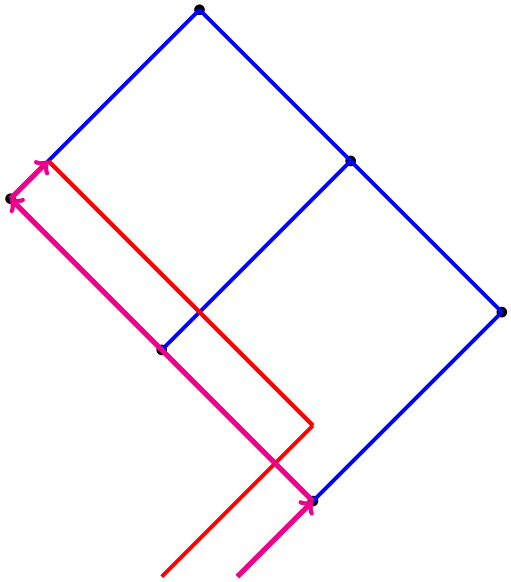

Figura 5.7: Caso en que la ruta es paralela a la rejilla

Al mismo tiempo esta es una variante de Line TRP en donde las ventanas de tiempo son unitarias y con tiempo de procesamiento para las tareas de 0. La decisión de colocar este tema en este capítulo es en primer lugar para mostrar que el problema máx COMSR es NP Duro y CTOMSR es NP Completo puesto que estos dos últimos son una generalización de Line TRP con tiempos de procesamiento 0, ya que Line TRP es el caso en que todas las $v_i = 0$ en CTOMSR y se sabe que dicha versión de Line TRP es NP Completa [16].

Para nuestro algoritmo retomaremos el algoritmo 3.1, puesto que en esencia se basa en el mismo principio de generar una cuadrícula de tamaño unitario sobre los segmentos ya rotados 45 grados. Se sabe que dada la cuadrícula unitaria y los segmentos, con las intersecciones de la rejilla como los nodos y el segmento de línea que los une como una arco, que en el caso horizontal tiene dirección izquierda a derecha y en el caso vertical tiene dirección de abajo hacia arriba, esta gráfica es un DAG. Si a cada arco de esta gráfica le colocamos como peso el número de objetos que intersecta, entonces es posible mediante un algoritmo de caminos más largos encontrar una ruta a través de la rejilla que intersecte al mayor número de objetos posibles, sin embargo, esto tiene un par de problemas: el primero quedó como el Lema 3.4 en donde se demuestra que un camino por la cuadrícula unitaria sólo intersectará a lo más a la cuarta parte de los segmentos que podria atrapar una ruta óptima en el plano. Por otro lado, el Lema 3.3 nos marca un error de conteo doble con respecto a los segmentos que intersectan la rejilla como se puede ver en la Figura 5.8, debido a su longitud y la manera en la que reside en la rejilla, toca a lo más 2 segmentos de esta.

Dadas estas condiciones, tenemos dos opciones. La primera es mejorar el factor de

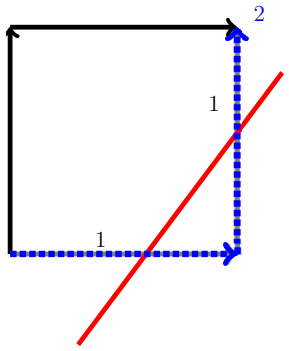

Figura 5.8: Ejemplo de error de conteo doble para un sólo segmento, la línea punteada azul es una ruta a través de la rejilla

aproximación de la rejilla y la segunda es evitar de alguna manera el conteo doble. En este caso mostraremos cómo es posible evitar el doble conteo. Para ello nos auxiliaremos de una gráfica dirigida $G' = (V', A')$. Por cada segmento horizontal y por cada segmento vertical de la rejilla unitaria crearemos un vértice, es decir $V' = V'_h \cup V'_v$ donde V_h son los vértices que corresponden a segmentos horizontales y V_v los de segmentos verticales. Por otro lado se crean arcos que unan los vértices horizontales cuyos segmentos correspondientes el la rejilla unitaria sean adyacentes (A'_h), arcos que unan los vértices verticales cuyos segmentos correspondientes en la rejilla unitaria sean adyacentes (A'_v) y otros que unan un vertical y un horizontal cuyos segmentos sean incidentes en la rejilla unitaria (A'_{vh}) y un horizontal con un vertical cuyos segmentos sean incidentes en la rejilla unitaria (A'_{hv}). $A' = A'_h \cup A'_v \cup A'_{hv} \cup A'_{vh}$.

Algoritmo 5.1 Algoritmo con factor de aproximación 4 para Line-TRP con ventanas de tiempo unitarias

Entrada: Un conjunto S de n segmentos unitarios, cuadrícula unitaria
Salida: Caminos más largos en G'

1: Se crea la gráfica dirigida $G' = (V', A')$
2: A cada arco de A' se le asigna el número de segmentos que lo cruzan.
3: Calcular caminos más largos sobre $G' = (V', A')$
4: **devolver** Camino de costo máximo de G'

.

Teorema 5.3 *El algoritmo 5.1 calcula de manera exacta el camino sobre la cuadrícula que intersecta a la máxima cantidad de segmentos.*

Demostración. Cada segmento unitario corta exactamente a tres arcos de G' como se ve en la Figura 5.10, pero de esos tres arcos no se pueden seguir dos en la misma ruta. De ser

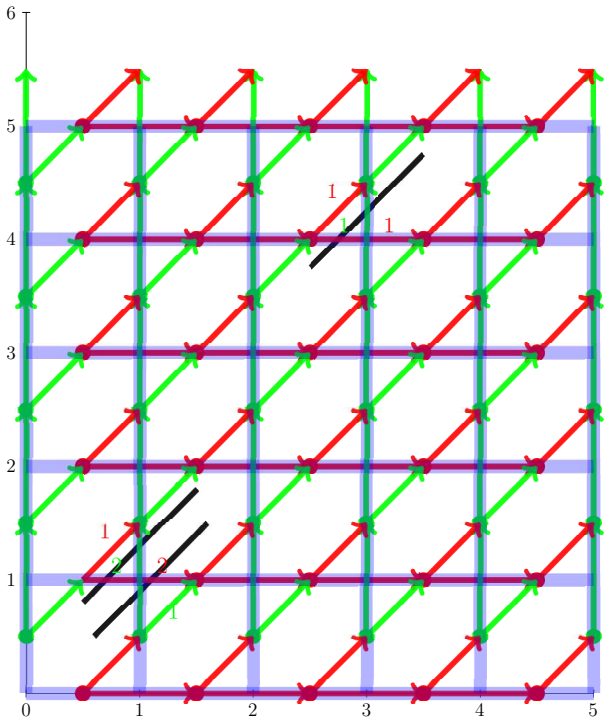

Figura 5.9: Ejemplo de la gráfica generada superpuesta en la cuadrícula original. Se muestra V'_h de rojo, V'_v de verde, así como A'_h de rojo, A'_v de verde, A'_{hv} de rojo y A'_{vh} de verde. Por último vemos la contabilización los arcos que tocan algún segmento.

así implicaría que el agente puede retroceder en el tiempo puesto que sería necesario un arco de regreso. ■

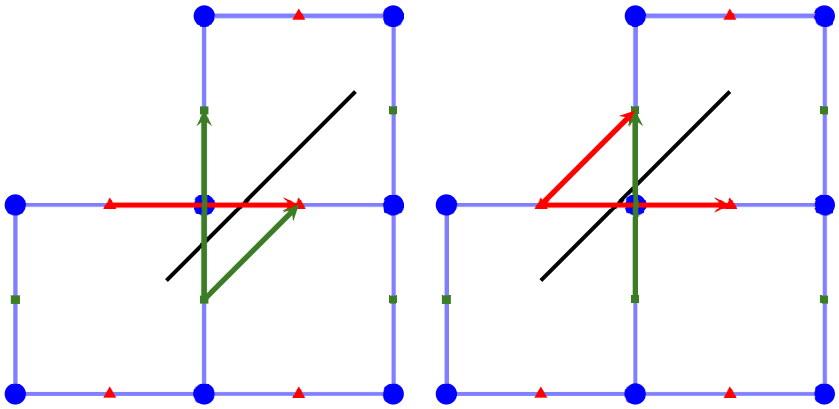

Figura 5.10: Se puede observar que cada vez que un segmento intersecta a la cuadrícula dos veces, sólo hay 3 arcos involucrados, el color azul es la cuadrícula original mientras que lo rojo representa a V'_h A'_h y A'_{hv} y lo verde a V'_v, A'_v, A'_{vh}.

Teorema 5.4 *El algoritmo 5.1 tiene un factor de aproximación 4*

Demostración. Se sabe por el Lema 3.4 que la cuadrícula original tiene un factor de aproximación 4. Puesto que la gráfica G' codifica todos los posibles caminos que puede seguir el agente sobre la cuadrícula original sin permitir duplicados. Por el Lema 5.3 el factor de aproximación del algoritmo 5.1 es 4. ■

Teorema 5.5 *El algoritmo 5.1 tiene tiempo de ejecución $O(n^2)$*

Demostración. Basados en el mismo argumento del algoritmo de factor de aproximación 8 expuesto en el capitulo 3, se sabe que la rejilla unitaria es de un tamaño $O(n^2)$ el numero de arcos en dicha rejilla es también de orden $O(n^2)$ lo que implica que $|V'|$ es de orden $O(n^2)$ y como de cada uno de estos vértices salen dos arcos, entonces el numero de arcos es también $O(n^2)$. Por lo tanto la construcción de G' requiere tiempo $O(n^2)$. La asignación de pesos a los arcos solo depende del numero de segmentos unitarios y el calculo de caminos más largos sobre G' al ser un DAG se puede hacer en tiempo $O(V' + A')$ como ya sabemos de que orden son V' y A' entonces se necesitan $O(n^2)$ pasos para este calculo. ■

Capítulo 6
Conclusiones y trabajo futuro

En este trabajo se comenzó estudiando el problema de captura de objetos sobre una recta de una generalización como una variante del agente viajero con objetivos móviles. A pesar de ser abordado mediante programación lineal, el poder de esta herramienta sólo sirvió en los problemas que implicaban orden, es así como se recurrió a la programación dinámica mediante la discretización del plano, ello bastó para demostrar que dicho mecanismo es suficiente para encontrar una solución óptima en un espacio finito de soluciones. La complejidad del problema no pudo ser disminuida siguiendo este camino. Es aquí donde el problema del reparador juega un papel importarte al ser éste último un caso especial del problema de este trabajo y al estar demostrado que este problema está en el conjunto de problemas NP-Completos. Se tuvo una contribución en lo que respecta a este problema con ventanas de tiempo unitarias, ya que se mejoraron dos algoritmos del estado del avance del conocimiento, en cuanto a su factor de aproximación y se encontró una mejora en cuanto a tiempo de ejecución con respecto a otros 2 algoritmos.

En cuanto al trabajo futuro en esta línea de investigación se podrían contemplar las siguientes propuestas:

- Mejorar el factor de aproximación para Line-TRP con ventanas de tiempo unitarias.
- Usar el algoritmo de Line-TRP con ventanas unitarias para CTOMSR y sus variantes.
- Modelar mediante programación entera Line-TRP y CTOMSR.
- Encontrar la complejidad para Line-TRP con ventanas unitarias.

Bibliografía

[1] Applegate D. L., et. al., *The Traveling Salesman Problem. A Computational Study*, Princeton University Press, 2006.

[2] Arora S., *Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and Other Geometric Problems*, ACM, 1998.

[3] Asahiro Y., et. al., *How to collect balls moving in the Euclidean plane*, Discrete Applied Mathematics, 154 (2006), pp. 2247 - 2262.

[4] Bar-Yehuda R., et. al., *On approximating a geometric prize-collecting traveling salesman problem with time windows*, Journal of Algorithms, 55(2005), pp. 76–92.

[5] Bellman R., Kalaba R., *Dynamic programming and modern control theory*, Academic Press(1965), pp. 195-209.

[6] Christofides N., *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem*, CARNEGIE-MELLON UNIV PITTSBURGH PA MANAGEMENT SCIENCES RESEARCH GROUP, 1976, pp. 1-11.

[7] Dantzig G., et. al., *Solution of a Large-Scale Traveling-Salesman Problem*, Journal of the Operations Research Society of America, 2(1954), pp. 393-410.

[8] Frederickson G. N., Wittman B., *Approximation Algorithms for the Traveling Repairman and Speeding Deliveryman Problems*, Algorithmica (2012), Vol 62, pp. 1198–1221.

[9] Garey M. R., Johnson D. S., *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman Co, USA, 1990.

[10] Helvig C. S., et. al., *The moving-target traveling salesman problem*, Journal of Algorithms, 49 (2003), pp. 153–174.

[11] Khachiyan L. G., *Polynomial algorithms in linear programming*, USSR Computational Mathematics and Mathematical Physics, **20** (1980), 53–72.

[12] Karmarkar N., *A new polynomial-time algorithm for linear programming*, Combinatorica, **4** (1984), 373-395.

[13] Klee V., Minty G. J., *How good is the simplex algorithm*, Washington University Seattle Dept of Math, Technical report, (1970), pp. 30.

[14] Papadimitriou C. H., *The Euclidean travelling salesman problem is NP-complete*, Theoretical Computer Science, **4**, Elsevier (1977), 237-244.

[15] Papadimitriou C., *Combinatorial optimization: algorithms and complexity*, Dover, 1998.

[16] Tsitsiklis J. N. , *Special Cases of Traveling Salesman and Repairman Problems with Time Windows*, NETWORKS, Vol. 22 (1992), pp. 263-282.